CP/M 2.2 INTERFACE GUIDE

1. INTRODUCTION.

This manual describes CP/M, release 2, system organization including the structure of memory and system entry points. The intention is to provide the necessary information required to write programs which operate under CP/M, and which use the peripheral and disk I/O facilities of the system.

CP/M is logically divided into four parts, called the Basic I/O System (BIOS), the Basic Disk Operating System (BDOS), the Console command processor (CCP), and the Transient Program Area (TPA). The BIOS is a hardware-dependent module which defines the exact low level interface to a particular computer system which is necessary for peripheral device I/O. Although a standard BIOS is supplied by Digital Research, explicit instructions are provided for field reconfiguration of the BIOS to match nearly any hardware environment (see the Digital Research manual entitled "CP/M Alteration Guide"). The BIOS and BDOS are logically combined into a single module with a common entry point, and referred to as the FDOS. The CCP is a distinct program which uses the FDOS to provide a human-oriented interface to the information which is cataloged on the backup storage device. The TPA is an area of memory (i.e., the portion which is not used by the FDOS and CCP) where various non-resident operating system commands and user programs are executed. The lower portion of memory is reserved for system information and is detailed later sections. Memory organization of the CP/M system in shown below:

```
            ------------------------------
   high    |                              |
   memory  |                              |
           |      FDOS (BDOS+BIOS)        |
           |                              |
   FBASE:  |                              |
            ------------------------------
           |                              |
           |            CCP               |
   CBASE:  |                              |
            ------------------------------
           |                              |
           |                              |
           |                              |
           |            TPA               |
           |                              |
   TBASE:  |                              |
            ------------------------------
           |     system  parameters       |
   BOOT:   |                              |
            ------------------------------
```

The exact memory addresses corresponding to BOOT, TBASE, CBASE, and FBASE vary from version to version, and are described fully in the "CP/M Alteration Guide." All standard CP/M versions, however, assume BOOT = 0000H, which is the base of random access memory. The machine code found at location BOOT performs a system "warm start" which loads and initializes the programs and variables necessary to return control to the CCP. Thus, transient programs need only jump to location BOOT

to return control to CP/M at the command level. Further, the standard versions assume TBASE = BOOT+0100H which is normally location 0100H. The principal entry point to the FDOS is at location BOOT+0005H (normally 0005H) where a jump to FBASE is found. The address field at BOOT+0006H (normally 0006H) contains the value of FBASE and can be used to determine the size of available memory, assuming the CCP is being overlayed by a transient program.

Transient programs are loaded into the TPA and executed as follows. The operator communicates with the CCP by typing command lines following each prompt. Each command line takes one of the forms:

```
command
command file1
command file1 file2
```

where "command" is either a built-in function such as DIR or TYPE, or the name of a transient command or program. If the command is a built-in function of CP/M, it is executed immediately. Otherwise, the CCP searches the currently addressed disk for a file by the name

```
command.COM
```

If the file is found, it is assumed to be a memory image of a program which executes in the TPA, and thus implicitly originates at TBASE in memory. The CCP loads the COM file from the disk into memory starting at TBASE and possibly extending up to CBASE.

If the command is followed by one or two file specifications, the CCP prepares one or two file control block (FCB) names in the system parameter area. These optional FCB's are in the form necessary to access files through the FDOS, and are described in the next section.

The transient program receives control from the CCP and begins execution, perhaps using the I/O facilities of the FDOS. The transient program is "called" from the CCP, and thus can simply return to the CCP upon completion of its processing, or can jump to BOOT to pass control back to CP/M. In the first case, the transient program must not use memory above CBASE, while in the latter case, memory up through FBASE-1 is free.

The transient program may use the CP/M I/O facilities to communicate with the operator's console and peripheral devices, including the disk subsystem. The I/O system is accessed by passing a "function number" and an "information address" to CP/M through the FDOS entry point at BOOT+0005H. In the case of a disk read, for example, the transient program sends the number corresponding to a disk read, along with the address of an FCB to the CP/M FDOS. The FDOS, in turn, performs the operation and returns with either a disk read completion indication or an error number indicating that the disk read was unsuccessful. The function numbers and error indicators are given in below.

2. OPERATING SYSTEM CALL CONVENTIONS.

The purpose of this section is to provide detailed information for performing direct operating system calls from user programs. Many of the functions listed below, however, are more simply accessed through the I/O macro library provided with the MAC macro assembler, and listed in the Digital Research manual entitled "MAC Macro Assembler: Language Manual and Applications Guide."

CP/M facilities which are available for access by transient programs fall into two general categories: simple device I/O, and disk file I/O. The simple device operations include:

- Read a Console Character
- Write a Console Character
- Read a Sequential Tape Character
- Write a Sequential Tape Character
- Write a List Device Character
- Get or Set I/O Status
- Print Console Buffer
- Read Console Buffer
- Interrogate Console Ready

The FDOS operations which perform disk Input/Output are

- Disk System Reset
- Drive Selection
- File Creation
- File Open
- File Close
- Directory Search
- File Delete
- File Rename
- Random or Sequential Read
- Random or Sequential Write
- Interrogate Available Disks
- Interrogate Selected Disk
- Set DMA Address
- Set/Reset File Indicators

As mentioned above, access to the FDOS functions is accomplished by passing a function number and information address through the primary entry point at location BOOT+0005H. In general, the function number is passed in register C with the information address in the double byte pair DE. Single byte values are returned in register A, with double byte values returned in HL (a zero value is returned when the function number is out of range). For reasons of compatibility, register A = L and register B = H upon return in all cases. Note that the register passing conventions of CP/M agree with those of Intel's PL/M systems programming language. The list of CP/M function numbers is given below.

0	System Reset	19	Delete File
1	Console Input	20	Read Sequential
2	Console Output	21	Write Sequential
3	Reader Input	22	Make File
4	Punch Output	23	Rename File
5	List Output	24	Return Login Vector
6	Direct Console I/O	25	Return Current Disk
7	Get I/O Byte	26	Set DMA Address
8	Set I/O Byte	27	Get Addr(Alloc)
9	Print String	28	Write Protect Disk
10	Read Console Buffer	29	Get R/O Vector
11	Get Console Status	30	Set File Attributes
12	Return Version Number	31	Get Addr(Disk Parms)
13	Reset Disk System	32	Set/Get User Code
14	Select Disk	33	Read Random
15	Open File	34	Write Random
16	Close File	35	Compute File Size
17	Search for First	36	Set Random Record
18	Search for Next		

(Functions 28 and 32 should be avoided in application programs to maintain upward compatibility with MP/M.)

Upon entry to a transient program, the CCP leaves the stack pointer set to an eight level stack area with the CCP return address pushed onto the stack, leaving seven levels before overflow occurs. Although this stack is usually not used by a transient program (i.e., most transients return to the CCP though a jump to location 0000H), it is sufficiently large to make CP/M system calls since the FDOS switches to a local stack at system entry. The following assembly language program segment, for example, reads characters continuously until an asterisk is encountered, at which time control returns to the CCP (assuming a standard CP/M system with BOOT = 0000H):

```
BDOS    EQU     0005H       ;STANDARD CP/M ENTRY
CONIN   EQU     1           ;CONSOLE INPUT FUNCTION
;
        ORG     0100H       ;BASE OF TPA
NEXTC:  MVI     C,CONIN     ;READ NEXT CHARACTER
        CALL    BDOS        ;RETURN CHARACTER IN <A>
        CPI     '*'         ;END OF PROCESSING?
        JNZ     NEXTC       ;LOOP IF NOT
        RET                 ;RETURN TO CCP
        END
```

CP/M implements a named file structure on each disk, providing a logical organization which allows any particular file to contain any number of records from completely empty, to the full capacity of the drive. Each drive is logically distinct with a disk directory and file data area. The disk file names are in three parts: the drive select code, the file name consisting of one to eight non-blank characters, and the file type consisting of zero to three non-blank characters. The file type names the generic category of a particular file, while the file name distinguishes individual files in each category. The file types listed below name a few generic categories

which have been established, although they are generally arbitrary:

ASM	Assembler Source	PLI	PL/I Source File
PRN	Printer Listing	REL	Relocatable Module
HEX	Hex Machine Code	TEX	TEX Formatter Source
BAS	Basic Source File	BAK	ED Source Backup
INT	Intermediate Code	SYM	SID Symbol File
COM	CCP Command File	$$$	Temporary File

Source files are treated as a sequence of ASCII characters, where each "line" of the source file is followed by a carriage-return line-feed sequence (ØDH followed by ØAH). Thus one 128 byte CP/M record could contain several lines of source text. The end of an ASCII file is denoted by a control-Z character (1AH) or a real end of file, returned by the CP/M read operation. Control-Z characters embedded within machine code files (e.g., COM files) are ignored, however, and the end of file condition returned by CP/M is used to terminate read operations.

Files in CP/M can be thought of as a sequence of up to 65536 records of 128 bytes each, numbered from Ø through 65535, thus allowing a maximum of 8 megabytes per file. Note, however, that although the records may be considered logically contiguous, they may not be physically contiguous in the disk data area. Internally, all files are broken into 16K byte segments called logical extents, so that counters are easily maintained as 8-bit values. Although the decomposition into extents is discussed in the paragraphs which follow, they are of no particular consequence to the programmer since each extent is automatically accessed in both sequential and random access modes.

In the file operations starting with function number 15, DE usually addresses a file control block (FCB). Transient programs often use the default file control block area reserved by CP/M at location BOOT+ØØ5CH (normally ØØ5CH) for simple file operations. The basic unit of file information is a 128 byte record used for all file operations, thus a default location for disk I/O is provided by CP/M at location BOOT+ØØ8ØH (normally ØØ8ØH) which is the initial default DMA address (see function 26). All directory operations take place in a reserved area which does not affect write buffers as was the case in release 1, with the exception of Search First and Search Next, where compatibility is required.

The File Control Block (FCB) data area consists of a sequence of 33 bytes for sequential access and a series of 36 bytes in the case that the file is accessed randomly. The default file control block normally located at ØØ5CH can be used for random access files, since the three bytes starting at BOOT+ØØ7DH are available for this purpose. The FCB format is shown with the following fields:

```
-----------------------------------------------------------
|dr|f1|f2|/ /|f8|t1|t2|t3|ex|s1|s2|rc|d0|/ /|dn|cr|r0|r1|r2|
-----------------------------------------------------------
 00 01 02 ... 08 09 10 11 12 13 14 15 16 ... 31 32 33 34 35
```

where

```
   dr         drive code (0 - 16)
              0 => use default drive for file
              1 => auto disk select drive A,
              2 => auto disk select drive B,
              ...
              16=> auto disk select drive P.

   f1...f8    contain the file name in ASCII
              upper case, with high bit = 0

   t1,t2,t3   contain the file type in ASCII
              upper case, with high bit = 0
              t1', t2', and t3' denote the
              bit of these positions,
              t1' = 1 => Read/Only file,
              t2' = 1 => SYS file, no DIR list

   ex         contains the current extent number,
              normally set to 00 by the user, but
              in range 0 - 31 during file I/O

   s1         reserved for internal system use

   s2         reserved for internal system use, set
              to zero on call to OPEN, MAKE, SEARCH

   rc         record count for extent "ex,"
              takes on values from 0 - 128

   d0...dn    filled-in by CP/M, reserved for
              system use

   cr         current record to read or write in
              a sequential file operation, normally
              set to zero by user

   r0,r1,r2   optional random record number in the
              range 0-65535, with overflow to r2,
              r0,r1 constitute a 16-bit value with
              low byte r0, and high byte r1
```

Each file being accessed through CP/M must have a corresponding FCB which provides the name and allocation information for all subsequent file operations. When accessing files, it is the programmer's responsibility to fill the lower sixteen bytes of the FCB and initialize the "cr" field. Normally, bytes 1 through 11 are set to the ASCII character values for the file name and file type, while all other fields are zero.

FCB's are stored in a directory area of the disk, and are brought into central memory before proceeding with file operations (see the OPEN and MAKE functions). The memory copy of the FCB is updated as file operations take place and later recorded permanently on disk at the termination of the file operation (see the CLOSE command).

The CCP constructs the first sixteen bytes of two optional FCB's for a transient by scanning the remainder of the line following the transient name, denoted by "file1" and "file2" in the prototype command line described above, with unspecified fields set to ASCII blanks. The first FCB is constructed at location BOOT+005CH, and can be used as-is for subsequent file operations. The second FCB occupies the d0 ... dn portion of the first FCB, and must be moved to another area of memory before use. If, for example, the operator types

PROGNAME B:X.ZOT Y.ZAP

the file PROGNAME.COM is loaded into the TPA, and the default FCB at BOOT+005CH is initialized to drive code 2, file name "X" and file type "ZOT". The second drive code takes the default value 0, which is placed at BOOT+006CH, with the file name "Y" placed into location BOOT+006DH and file type "ZAP" located 8 bytes later at BOOT+0075H. All remaining fields through "cr" are set to zero. Note again that it is the programmer's responsibility to move this second file name and type to another area, usually a separate file control block, before opening the file which begins at BOOT+005CH, due to the fact that the open operation will overwrite the second name and type.

If no file names are specified in the original command, then the fields beginning at BOOT+005DH and BOOT+006DH contain blanks. In all cases, the CCP translates lower case alphabetics to upper case to be consistent with the CP/M file naming conventions.

As an added convenience, the default buffer area at location BOOT+0080H is initialized to the command line tail typed by the operator following the program name. The first position contains the number of characters, with the characters themselves following the character count. Given the above command line, the area beginning at BOOT+0080H is initialized as follows:

```
BOOT+0080H:
+00 +01 +02 +03 +04 +05 +06 +07 +08 +09 +10 +11 +12 +13 +14
 14 " " "B" ":" "X" "." "Z" "O" "T" " " "Y" "." "Z" "A" "P"
```

where the characters are translated to upper case ASCII with uninitialized memory following the last valid character. Again, it is the responsibility of the programmer to extract the information from this buffer before any file operations are performed, unless the default DMA address is explicitly changed.

The individual functions are described in detail in the pages which follow.

```
****************************************
*                                      *
*  FUNCTION Ø:  System Reset           *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  ØØH              *
****************************************
```

The system reset function returns control to the CP/M operating system at the CCP level. The CCP re-initializes the disk subsystem by selecting and logging-in disk drive A. This function has exactly the same effect as a jump to location BOOT.

```
****************************************
*                                      *
*  FUNCTION 1:  CONSOLE INPUT          *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  Ø1H              *
*                                      *
*  Returned   Value:                   *
*      Register   A:  ASCII Character  *
****************************************
```

The console input function reads the next console character to register A. Graphic characters, along with carriage return, line feed, and backspace (ctl-H) are echoed to the console. Tab characters (ctl-I) are expanded in columns of eight characters. A check is made for start/stop scroll (ctl-S) and start/stop printer echo (ctl-P). The FDOS does not return to the calling program until a character has been typed, thus suspending execution if a character is not ready.

```
****************************************
*                                      *
*  FUNCTION 2:  CONSOLE OUTPUT         *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  Ø2H              *
*      Register   E:  ASCII Character  *
*                                      *
****************************************
```

The ASCII character from register E is sent to the console device. Similar to function 1, tabs are expanded and checks are made for start/stop scroll and printer echo.

```
*****************************************
*                                       *
*  FUNCTION 3:  READER INPUT            *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  Ø3H               *
*                                       *
*  Returned   Value:                    *
*      Register   A:  ASCII Character   *
*****************************************
```

The Reader Input function reads the next character from the logical reader into register A (see the IOBYTE definition in the "CP/M Alteration Guide"). Control does not return until the character has been read.

```
*****************************************
*                                       *
*  FUNCTION 4:  PUNCH OUTPUT            *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  Ø4H               *
*      Register   E:  ASCII Character   *
*                                       *
*****************************************
```

The Punch Output function sends the character from register E to the logical punch device.

```
*****************************************
*                                       *
*  FUNCTION 5:  LIST OUTPUT             *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  Ø5H               *
*      Register   E:  ASCII Character   *
*                                       *
*****************************************
```

The List Output function sends the ASCII character in register E to the logical listing device.

```
*****************************************
*                                       *
*  FUNCTION 6:  DIRECT CONSOLE I/O      *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  06H               *
*      Register   E:  0FFH (input) or   *
*                     char (output)     *
*                                       *
*  Returned   Value:                    *
*      Register   A:  char or status    *
*                     (no value)        *
*****************************************
```

Direct console I/O is supported under CP/M for those specialized applications where unadorned console input and output is required. Use of this function should, in general, be avoided since it bypasses all of CP/M's normal control character functions (e.g., control-S and control-P). Programs which perform direct I/O through the BIOS under previous releases of CP/M, however, should be changed to use direct I/O under BDOS so that they can be fully supported under future releases of MP/M and CP/M.

Upon entry to function 6, register E either contains hexadecimal FF, denoting a console input request, or register E contains an ASCII character. If the input value is FF, then function 6 returns A = 00 if no character is ready, otherwise A contains the next console input character.

If the input value in E is not FF, then function 6 assumes that E contains a valid ASCII character which is sent to the console.

```
*****************************************
*                                       *
*  FUNCTION 7:  GET I/O BYTE            *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  Ø7H               *
*                                       *
*  Returned   Value:                    *
*      Register   A:  I/O Byte Value    *
*****************************************
```

The Get I/O Byte function returns the current value of IOBYTE in register A. See the "CP/M Alteration Guide" for IOBYTE definition.

```
*****************************************
*                                       *
*  FUNCTION 8:  SET I/O BYTE            *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  Ø8H               *
*      Register   E:  I/O Byte Value    *
*                                       *
*****************************************
```

The Set I/O Byte function changes the system IOBYTE value to that given in register E.

```
*****************************************
*                                       *
*  FUNCTION 9:  PRINT STRING            *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  Ø9H               *
*      Registers DE:  String Address    *
*                                       *
*****************************************
```

The Print String function sends the character string stored in memory at the location given by DE to the console device, until a "$" is encountered in the string. Tabs are expanded as in function 2, and checks are made for start/stop scroll and printer echo.

```
****************************************
*                                      *
*  FUNCTION 10: READ CONSOLE BUFFER    *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  0AH              *
*      Registers DE:  Buffer Address   *
*                                      *
*  Returned   Value:                   *
*      Console Characters in Buffer    *
****************************************
```

The Read Buffer function reads a line of edited console input into a buffer addressed by registers DE. Console input is terminated when either the input buffer overflows. The Read Buffer takes the form:

```
DE: +0 +1 +2 +3 +4 +5 +6 +7 +8     . . .    +n
   -------------------------------------------
   |mx|nc|c1|c2|c3|c4|c5|c6|c7|    . . .   |??|
   -------------------------------------------
```

where "mx" is the maximum number of characters which the buffer will hold (1 to 255), "nc" is the number of characters read (set by FDOS upon return), followed by the characters read from the console. if nc < mx, then uninitialized positions follow the last character, denoted by "??" in the above figure. A number of control functions are recognized during line editing:

```
rub/del removes and echoes the last character
  ctl-C  reboots when at the beginning of line
  ctl-E  causes physical end of line
  ctl-H  backspaces one character position
  ctl-J  (line feed) terminates input line
  ctl-M  (return) terminates input line
  ctl-R  retypes the current line after new line
  ctl-U  removes currnt line after new line
  ctl-X  backspaces to beginning of current line
```

Note also that certain functions which return the carriage to the leftmost position (e.g., ctl-X) do so only to the column position where the prompt ended (in earlier releases, the carriage returned to the extreme left margin). This convention makes operator data input and line correction more legible.

```
*****************************************
*                                       *
*  FUNCTION 11: GET CONSOLE STATUS      *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  ØBH               *
*                                       *
*  Returned   Value:                    *
*      Register   A:  Console Status    *
*****************************************
```

The Console Status function checks to see if a character has been typed at the console. If a character is ready, the value ØFFH is returned in register A. Otherwise a ØØH value is returned.

```
*****************************************
*                                       *
*  FUNCTION 12: RETURN VERSION NUMBER   *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  ØCH               *
*                                       *
*  Returned   Value:                    *
*      Registers HL:  Version Number    *
*****************************************
```

Function 12 provides information which allows version independent programming. A two-byte value is returned, with H = ØØ designating the CP/M release (H = Ø1 for MP/M), and L = ØØ for all releases previous to 2.Ø. CP/M 2.Ø returns a hexadecimal 2Ø in register L, with subsequent version 2 releases in the hexadecimal range 21, 22, through 2F. Using function 12, for example, you can write application programs which provide both sequential and random access functions, with random access disabled when operating under early releases of CP/M.

```
****************************************
*                                      *
*  FUNCTION 13: RESET DISK SYSTEM      *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  0DH              *
*                                      *
****************************************
```

The Reset Disk Function is used to programmatically restore the file system to a reset state where all disks are set to read/write (see functions 28 and 29), only disk drive A is selected, and the default DMA address is reset to BOOT+0080H. This function can be used, for example, by an application program which requires a disk change without a system reboot.

```
****************************************
*                                      *
*  FUNCTION 14: SELECT DISK            *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  0EH              *
*      Register   E:  Selected Disk    *
*                                      *
****************************************
```

The Select Disk function designates the disk drive named in register E as the default disk for subsequent file operations, with E = 0 for drive A, 1 for drive B, and so-forth through 15 corresponding to drive P in a full sixteen drive system. The drive is placed in an "on-line" status which, in particular, activates its directory until the next cold start, warm start, or disk system reset operation. If the disk media is changed while it is on-line, the drive automatically goes to a read/only status in a standard CP/M environment (see function 28). FCB's which specify drive code zero (dr = 00H) automatically reference the currently selected default drive. Drive code values between 1 and 16, however, ignore the selected default drive and directly reference drives A through P.

```
*****************************************
*                                       *
*  FUNCTION 15: OPEN FILE               *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  ØFH               *
*      Registers DE:  FCB Address       *
*                                       *
*  Returned   Value:                    *
*      Register   A:  Directory Code    *
*****************************************
```

The Open File operation is used to activate a file which currently exists in the disk directory for the currently active user number. The FDOS scans the referenced disk directory for a match in positions 1 through 14 of the FCB referenced by DE (byte s1 is automatically zeroed), where an ASCII question mark (3FH) matches any directory character in any of these positions. Normally, no question marks are included and, further, bytes "ex" and "s2" of the FCB are zero.

If a directory element is matched, the relevant directory information is copied into bytes dØ through dn of the FCB, thus allowing access to the files through subsequent read and write operations. Note that an existing file must not be accessed until a sucessful open operation is completed. Upon return, the open function returns a "directory code" with the value Ø through 3 if the open was successful, or ØFFH (255 decimal) if the file cannot be found. If question marks occur in the FCB then the first matching FCB is activated. Note that the current record ("cr") must be zeroed by the program if the file is to be accessed sequentially from the first record.

```
***************************************
*                                     *
*  FUNCTION 16: CLOSE FILE            *
*                                     *
***************************************
*  Entry Parameters:                  *
*      Register   C:  10H             *
*      Registers DE:  FCB Address     *
*                                     *
*  Returned   Value:                  *
*      Register   A:  Directory Code  *
***************************************
```

The Close File function performs the inverse of the open file function. Given that the FCB addressed by DE has been previously activated through an open or make function (see functions 15 and 22), the close function permanently records the new FCB in the referenced disk directory. The FCB matching process for the close is identical to the open function. The directory code returned for a successful close operation is 0, 1, 2, or 3, while a 0FFH (255 decimal) is returned if the file name cannot be found in the directory. A file need not be closed if only read operations have taken place. If write operations have occurred, however, the close operation is necessary to permanently record the new directory information.

```
****************************************
*                                      *
*  FUNCTION 17: SEARCH FOR FIRST       *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register    C:  11H             *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

Search First scans the directory for a match with the file given by the FCB addressed by DE. The value 255 (hexadecimal FF) is returned if the file is not found, otherwise Ø, 1, 2, or 3 is returned indicating the file is present. In the case that the file is found, the current DMA address is filled with the record containing the directory entry, and the relative starting position is A * 32 (i.e., rotate the A register left 5 bits, or ADD A five times). Although not normally required for application programs, the directory information can be extracted from the buffer at this position.

An ASCII question mark (63 decimal, 3F hexadecimal) in any position from "f1" through "ex" matches the corresponding field of any directory entry on the default or auto-selected disk drive. If the "dr" field contains an ASCII question mark, then the auto disk select function is disabled, the default disk is searched, with the search function returning any matched entry, allocated or free, belonging to any user number. This latter function is not normally used by application programs, but does allow complete flexibility to scan all current directory values. If the "dr" field is not a question mark, the "s2" byte is automatically zeroed.

```
****************************************
*                                      *
*  FUNCTION 18: SEARCH FOR NEXT        *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  12H              *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

The Search Next function is similar to the Search First function, except that the directory scan continues from the last matched entry. Similar to function 17, function 18 returns the decimal value 255 in A when no more directory items match.

```
****************************************
*                                      *
*  FUNCTION 19: DELETE FILE            *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  13H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

The Delete File function removes files which match the FCB addressed by DE. The filename and type may contain ambiguous references (i.e., question marks in various positions), but the drive select code cannot be ambiguous, as in the Search and Search Next functions.

Function 19 returns a decimal 255 if the referenced file or files cannot be found, otherwise a value in the range 0 to 3 is returned.

```
****************************************
*                                      *
*  FUNCTION 20: READ SEQUENTIAL        *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  14H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

Given that the FCB addressed by DE has been activated through an open or make function (numbers 15 and 22), the Read Sequential function reads the next 128 byte record from the file into memory at the current DMA address. the record is read from position "cr" of the extent, and the "cr" field is automatically incremented to the next record position. If the "cr" field overflows then the next logical extent is automatically opened and the "cr" field is reset to zero in preparation for the next read operation. The value 00H is returned in the A register if the read operation was successful, while a non-zero value is returned if no data exists at the next record position (e.g., end of file occurs).

```
****************************************
*                                      *
*  FUNCTION 21: WRITE SEQUENTIAL       *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  15H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

Given that the FCb addressed by DE has been activated through an open or make function (numbers 15 and 22), the Write Sequential function writes the 128 byte data record at the current DMA address to the file named by the FCB. the record is placed at position "cr" of the file, and the "cr" field is automatically incremented to the next record position. If the "cr" field overflows then the next logical extent is automatically opened and the "cr" field is reset to zero in preparation for the next write operation. Write operations can take place into an existing file, in which case newly written records overlay those which already exist in the file. Register A = ØØH upon return from a successful write operation, while a non-zero value indicates an unsuccessful write due to a full disk.

```
****************************************
*                                      *
*  FUNCTION 22: MAKE FILE              *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  16H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

The Make File operation is similar to the open file operation except that the FCB must name a file which does not exist in the currently referenced disk directory (i.e., the one named explicitly by a non-zero "dr" code, or the default disk if "dr" is zero). The FDOS creates the file and initializes both the directory and main memory value to an empty file. The programmer must ensure that no duplicate file names occur, and a preceding delete operation is sufficient if there is any possibility of duplication. Upon return, register A = Ø, 1, 2, or 3 if the operation was successful and ØFFH (255 decimal) if no more directory space is available. The make function has the side-effect of activating the FCB and thus a subsequent open is not necessary.

```
*****************************************
*                                       *
*  FUNCTION 23: RENAME FILE             *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  17H               *
*      Registers DE:  FCB Address       *
*                                       *
*  Returned   Value:                    *
*      Register   A:  Directory Code    *
*****************************************
```

The Rename function uses the FCB addressed by DE to change all occurrences of the file named in the first 16 bytes to the file named in the second 16 bytes. The drive code "dr" at position 0 is used to select the drive, while the drive code for the new file name at position 16 of the FCB is assumed to be zero. Upon return, register A is set to a value between 0 and 3 if the rename was successful, and 0FFH (255 decimal) if the first file name could not be found in the directory scan.

```
*****************************************
*                                       *
*  FUNCTION 24: RETURN LOGIN VECTOR     *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  18H               *
*                                       *
*  Returned   Value:                    *
*      Registers HL:  Login Vector      *
*****************************************
```

The login vector value returned by CP/M is a 16-bit value in HL, where the least significant bit of L corresponds to the first drive A, and the high order bit of H corresponds to the sixteenth drive, labelled P. A "0" bit indicates that the drive is not on-line, while a "1" bit marks an drive that is actively on-line due to an explicit disk drive selection, or an implicit drive select caused by a file operation which specified a non-zero "dr" field. Note that compatibility is maintained with earlier releases, since registers A and L contain the same values upon return.

```
****************************************
*                                      *
*  FUNCTION 25: RETURN CURRENT DISK    *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  19H              *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Current Disk     *
****************************************
```

Function 25 returns the currently selected default disk number in register A. The disk numbers range from 0 through 15 corresponding to drives A through P.

```
****************************************
*                                      *
*  FUNCTION 26: SET DMA ADDRESS        *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  1AH              *
*      Registers DE:  DMA Address      *
*                                      *
****************************************
```

"DMA" is an acronym for Direct Memory Address, which is often used in connection with disk controllers which directly access the memory of the mainframe computer to transfer data to and from the disk subsystem. Although many computer systems use non-DMA access (i.e., the data is transfered through programmed I/O operations), the DMA address has, in CP/M, come to mean the address at which the 128 byte data record resides before a disk write and after a disk read. Upon cold start, warm start, or disk system reset, the DMA address is automatically set to BOOT+0080H. The Set DMA function, however, can be used to change this default value to address another area of memory where the data records reside. Thus, the DMA address becomes the value specified by DE until it is changed by a subsequent Set DMA function, cold start, warm start, or disk system reset.

```
****************************************
*                                      *
*  FUNCTION 27: GET ADDR(ALLOC)        *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  1BH              *
*                                      *
*  Returned   Value:                   *
*      Registers HL:  ALLOC Address    *
****************************************
```

An "allocation vector" is maintained in main memory for each on-line disk drive. Various system programs use the information provided by the allocation vector to determine the amount of remaining storage (see the STAT program). Function 27 returns the base address of the allocation vector for the currently selected disk drive. The allocation information may, however, be invalid if the selected disk has been marked read/only. Although this function is not normally used by application programs, additional details of the allocation vector are found in the "CP/M Alteration Guide."

```
****************************************
*                                      *
*  FUNCTION 28: WRITE PROTECT DISK     *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  1CH              *
*                                      *
****************************************
```

The disk write protect function provides temporary write protection for the currently selected disk. Any attempt to write to the disk, before the next cold or warm start operation produces the message

Bdos Err on d: R/O

```
****************************************
*                                      *
*  FUNCTION 29: GET READ/ONLY VECTOR   *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  1DH              *
*                                      *
*  Returned   Value:                   *
*      Registers HL:  R/O Vector Value*
****************************************
```

Function 29 returns a bit vector in register pair HL which indicates drives which have the temporary read/only bit set. Similar to function 24, the least significant bit corresponds to drive A, while the most significant bit corresponds to drive P. The R/O bit is set either by an explicit call to function 28, or by the automatic software mechanisms within CP/M which detect changed disks.

```
****************************************
*                                      *
*  FUNCTION 30: SET FILE ATTRIBUTES    *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  1EH              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

The Set File Attributes function allows programmatic manipulation of permanent indicators attached to files. In particular, the R/O and System attributes (t1' and t2') can be set or reset. The DE pair addresses an unambiguous file name with the appropriate attributes set or reset. Function 30 searches for a match, and changes the matched directory entry to contain the selected indicators. Indicators f1' through f4' are not presently used, but may be useful for applications programs, since they are not involved in the matching process during file open and close operations. Indicators f5' through f8' and t3' are reserved for future system expansion.

```
*****************************************
*                                       *
*  FUNCTION 31: GET ADDR(DISK PARMS)    *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  1FH               *
*                                       *
*  Returned   Value:                    *
*      Registers HL:  DPB Address       *
*****************************************
```

The address of the BIOS resident disk parameter block is returned in HL as a result of this function call. This address can be used for either of two purposes. First, the disk parameter values can be extracted for display and space computation purposes, or transient programs can dynamically change the values of current disk parameters when the disk environment changes, if required. Normally, application programs will not require this facility.

```
*****************************************
*                                       *
*  FUNCTION 32: SET/GET USER CODE       *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:  20H               *
*      Register   E:  0FFH (get) or     *
*                     User Code (set)   *
*                                       *
*  Returned   Value:                    *
*      Register   A:  Current Code or   *
*                     (no value)        *
*****************************************
```

An application program can change or interrogate the currently active user number by calling function 32. If register E = 0FFH, then the value of the current user number is returned in register A, where the value is in the range 0 to 31. If register E is not 0FFH, then the current user number is changed to the value of E (modulo 32).

```
****************************************
*                                      *
*  FUNCTION 33: READ RANDOM            *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  21H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Return Code      *
****************************************
```

The Read Random function is similar to the sequential file read operation of previous releases, except that the read operation takes place at a particular record number, selected by the 24-bit value constructed from the three byte field following the FCB (byte positions rØ at 33, r1 at 34, and r2 at 35). Note that the sequence of 24 bits is stored with least significant byte first (rØ), middle byte next (r1), and high byte last (r2). CP/M does not reference byte r2, except in computing the size of a file (function 35). Byte r2 must be zero, however, since a non-zero value indicates overflow past the end of file.

Thus, the rØ,r1 byte pair is treated as a double-byte, or "word" value, which contains the record to read. This value ranges from Ø to 65535, providing access to any particular record of the 8 megabyte file. In order to process a file using random access, the base extent (extent Ø) must first be opened. Although the base extent may or may not contain any allocated data, this ensures that the file is properly recorded in the directory, and is visible in DIR requests. The selected record number is then stored into the random record field (rØ,r1), and the BDOS is called to read the record. Upon return from the call, register A either contains an error code, as listed below, or the value ØØ indicating the operation was successful. In the latter case, the current DMA address contains the randomly accessed record. Note that contrary to the sequential read operation, the record number is not advanced. Thus, subsequent random read operations continue to read the same record.

Upon each random read operation, the logical extent and current record values are automatically set. Thus, the file can be sequentially read or written, starting from the current randomly accessed position. Note, however, that in this case, the last randomly read record will be re-read as you switch from random mode to sequential read, and the last record will be re-written as you switch to a sequential write operation. You can, of course, simply advance the random record position following each random read or write to obtain the effect of a sequential I/O operation.

Error codes returned in register A following a random read are listed below.

01	reading unwritten data
02	(not returned in random mode)
03	cannot close current extent
04	seek to unwritten extent
05	(not returned in read mode)
06	seek past physical end of disk

Error code 01 and 04 occur when a random read operation accesses a data block which has not been previously written, or an extent which has not been created, which are equivalent conditions. Error 3 does not normally occur under proper system operation, but can be cleared by simply re-reading, or re-opening extent zero as long as the disk is not physically write protected. Error code 06 occurs whenever byte r2 is non-zero under the current 2.0 release. Normally, non-zero return codes can be treated as missing data, with zero return codes indicating operation complete.

```
*****************************************
*                                       *
*  FUNCTION 34: WRITE RANDOM            *
*                                       *
*****************************************
*  Entry Parameters:                    *
*      Register   C:   22H              *
*      Registers DE:   FCB Address      *
*                                       *
*  Returned   Value:                    *
*      Register   A:   Return Code      *
*****************************************
```

The Write Random operation is initiated similar to the Read Random call, except that data is written to the disk from the current DMA address. Further, if the disk extent or data block which is the target of the write has not yet been allocated, the allocation is performed before the write operation continues. As in the Read Random operation, the random record number is not changed as a result of the write. The logical extent number and current record positions of the file control block are set to correspond to the random record which is being written. Again, sequential read or write operations can commence following a random write, with the notation that the currently addressed record is either read or rewritten again as the sequential operation begins. You can also simply advance the random record position following each write to get the effect of a sequential write operation. Note that in particular, reading or writing the last record of an extent in random mode does not cause an automatic extent switch as it does in sequential mode.

The error codes returned by a random write are identical to the random read operation with the addition of error code 05, which indicates that a new extent cannot be created due to directory overflow.

```
****************************************
*                                      *
*  FUNCTION 35: COMPUTE FILE SIZE      *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  23H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Random Record Field Set         *
****************************************
```

When computing the size of a file, the DE register pair addresses an FCB in random mode format (bytes r0, r1, and r2 are present). The FCB contains an unambiguous file name which is used in the directory scan. Upon return, the random record bytes contain the "virtual" file size which is, in effect, the record address of the record following the end of the file. if, following a call to function 35, the high record byte r2 is 01, then the file contains the maximum record count 65536. Otherwise, bytes r0 and r1 constitute a 16-bit value (r0 is the least significant byte, as before) which is the file size.

Data can be appended to the end of an existing file by simply calling function 35 to set the random record position to the end of file, then performing a sequence of random writes starting at the preset record address.

The virtual size of a file corresponds to the physical size when the file is written sequentially. If, instead, the file was created in random mode and "holes" exist in the allocation, then the file may in fact contain fewer records than the size indicates. If, for example, only the last record of an eight megabyte file is written in random mode (i.e., record number 65535), then the virtual size is 65536 records, although only one block of data is actually allocated.

```
****************************************
*                                      *
*  FUNCTION 36: SET RANDOM RECORD      *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  24H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Random Record Field Set         *
****************************************
```

The Set Random Record function causes the BDOS to automatically produce the random record position from a file which has been read or written sequentially to a particular point. The function can be useful in two ways.

First, it is often necessary to initially read and scan a sequential file to extract the positions of various "key" fields. As each key is encountered, function 36 is called to compute the random record position for the data corresponding to this key. If the data unit size is 128 bytes, the resulting record position is placed into a table with the key for later retrieval. After scanning the entire file and tabularizing the keys and their record numbers, you can move instantly to a particular keyed record by performing a random read using the corresponding random record number which was saved earlier. The scheme is easily generalized when variable record lengths are involved since the program need only store the buffer-relative byte position along with the key and record number in order to find the exact starting position of the keyed data at a later time.

A second use of function 36 occurs when switching from a sequential read or write over to random read or write. A file is sequentially accessed to a particular point in the file, function 36 is called which sets the record number, and subsequent random read and write operations continue from the selected point in the file.

3. A SAMPLE FILE-TO-FILE COPY PROGRAM.

The program shown below provides a relatively simple example of file operations. The program source file is created as COPY.ASM using the CP/M ED program and then assembled using ASM or MAC, resulting in a "HEX" file. The LOAD program is the used to produce a COPY.COM file which executes directly under the CCP. The program begins by setting the stack pointer to a local area, and then proceeds to move the second name from the default area at ØØ6CH to a 33-byte file control block called DFCB. The DFCB is then prepared for file operations by clearing the current record field. At this point, the source and destination FCB's are ready for processing since the SFCB at ØØ5CH is properly set-up by the CCP upon entry to the COPY program. That is, the first name is placed into the default fcb, with the proper fields zeroed, including the current record field at ØØ7CH. The program continues by opening the source file, deleting any exising destination file, and then creating the destination file. If all this is successful, the program loops at the label COPY until each record has been read from the source file and placed into the destination file. Upon completion of the data transfer, the destination file is closed and the program returns to the CCP command level by jumping to BOOT.

```
                ;       sample file-to-file copy program
                ;
                ;       at the ccp level, the command
                ;
                ;               copy a:x.y b:u.v
                ;
                ;       copies the file named x.y from drive
                ;       a to a file named u.v on drive b.
                ;
ØØØØ =          boot    equ     ØØØØh   ; system reboot
ØØØ5 =          bdos    equ     ØØØ5h   ; bdos entry point
ØØ5c =          fcbl    equ     ØØ5ch   ; first file name
ØØ5c =          sfcb    equ     fcbl    ; source fcb
ØØ6c =          fcb2    equ     ØØ6ch   ; second file name
ØØ8Ø =          dbuff   equ     ØØ8Øh   ; default buffer
Ø1ØØ =          tpa     equ     Ø1ØØh   ; beginning of tpa
                ;
ØØØ9 =          printf  equ     9       ; print buffer func#
ØØØf =          openf   equ     15      ; open file func#
ØØ1Ø =          closef  equ     16      ; close file func#
ØØ13 =          deletef equ     19      ; delete file func#
ØØ14 =          readf   equ     2Ø      ; sequential read
ØØ15 =          writef  equ     21      ; sequential write
ØØ16 =          makef   equ     22      ; make file func#
                ;
Ø1ØØ                    org     tpa     ; beginning of tpa
Ø1ØØ 311bØ2             lxi     sp,stack; local stack
                ;
                ;       move second file name to dfcb
Ø1Ø3 Øe1Ø               mvi     c,16    ; half an fcb
```

```
0105 116c00          lxi     d,fcb2  ; source of move
0108 21da01          lxi     h,dfcb  ; destination fcb
010b 1a      mfcb:   ldax    d       ; source fcb
010c 13              inx     d       ; ready next
010d 77              mov     m,a     ; dest fcb
010e 23              inx     h       ; ready next
010f 0d              dcr     c       ; count 16...0
0110 c20b01          jnz     mfcb    ; loop 16 times
             ;
             ;       name has been moved, zero cr
0113 af              xra     a       ; a = 00h
0114 32fa01          sta     dfcbcr  ; current rec = 0
             ;
             ;       source and destination fcb's ready
             ;
0117 115c00          lxi     d,sfcb  ; source file
011a cd6901          call    open    ; error if 255
011d 118701          lxi     d,nofile; ready message
0120 3c              inr     a       ; 255 becomes 0
0121 cc6101          cz      finis   ; done if no file
             ;
             ;       source file open, prep destination
0124 11da01          lxi     d,dfcb  ; destination
0127 cd7301          call    delete  ; remove if present
             ;
012a 11da01          lxi     d,dfcb  ; destination
012d cd8201          call    make    ; create the file
0130 119601          lxi     d,nodir ; ready message
0133 3c              inr     a       ; 255 becomes 0
0134 cc6101          cz      finis   ; done if no dir space
             ;
             ;       source file open, dest file open
             ;       copy until end of file on source
             ;
0137 115c00  copy:   lxi     d,sfcb  ; source
013a cd7801          call    read    ; read next record
013d b7              ora     a       ; end of file?
013e c25101          jnz     eofile  ; skip write if so
             ;
             ;       not end of file, write the record
0141 11da01          lxi     d,dfcb  ; destination
0144 cd7d01          call    write   ; write record
0147 11a901          lxi     d,space ; ready message
014a b7              ora     a       ; 00 if write ok
014b c46101          cnz     finis   ; end if so
014e c33701          jmp     copy    ; loop until eof
             ;
             eofile: ; end of file, close destination
0151 11da01          lxi     d,dfcb  ; destination
0154 cd6e01          call    close   ; 255 if error
0157 21bb01          lxi     h,wrprot; ready message
015a 3c              inr     a       ; 255 becomes 00
015b cc6101          cz      finis   ; shouldn't happen
             ;
             ;       copy operation complete, end
```

```
015e 11cc01            lxi     d,normal; ready message
               ;
               finis:  ; write message given by de, reboot
0161 0e09              mvi     c,printf
0163 cd0500            call    bdos    ; write message
0166 c30000            jmp     boot    ; reboot system
               ;
               ;       system interface subroutines
               ;       (all return directly from bdos)
               ;
0169 0e0f      open:   mvi     c,openf
016b c30500            jmp     bdos
               ;
016e 0e10      close:  mvi     c,closef
0170 c30500            jmp     bdos
               ;
0173 0e13      delete: mvi     c,deletef
0175 c30500            jmp     bdos
               ;
0178 0e14      read:   mvi     c,readf
017a c30500            jmp     bdos
               ;
017d 0e15      write:  mvi     c,writef
017f c30500            jmp     bdos
               ;
0182 0e16      make:   mvi     c,makef
0184 c30500            jmp     bdos
               ;
               ;       console messages
0187 6e6f20fnofile: db         'no source file$'
0196 6e6f209nodir:  db         'no directory space$'
01a9 6f7574fspace:  db         'out of data space$'
01bb 7772695wrprot: db         'write protected?$'
01cc 636f700normal: db         'copy complete$'
               ;
               ;       data areas
01da           dfcb:   ds      33         ; destination fcb
01fa =         dfcbcr  equ     dfcb+32 ; current record
               ;
01fb                   ds      32         ; 16 level stack
               stack:
021b                   end
```

Note that there are several simplifications in this particular program. First, there are no checks for invalid file names which could, for example, contain ambiguous references. This situation could be detected by scanning the 32 byte default area starting at location 005CH for ASCII question marks. A check should also be made to ensure that the file names have, in fact, been included (check locations 005DH and 006DH for non-blank ASCII characters). Finally, a check should be made to ensure that the source and destination file names are different. A speed improvement could be made by buffering more data on each read operation. One could, for example, determine

the size of memory by fetching FBASE from location 0006H and use the entire remaining portion of memory for a data buffer. In this case, the programmer simply resets the DMA address to the next successive 128 byte area before each read. Upon writing to the destination file, the DMA address is reset to the beginning of the buffer and incremented by 128 bytes to the end as each record is transferred to the destination file.

4. A SAMPLE FILE DUMP UTILITY.

The file dump program shown below is slightly more complex than the simple copy program given in the previous section. The dump program reads an input file, specified in the CCP command line, and displays the content of each record in hexadecimal format at the console. Note that the dump program saves the CCP's stack upon entry, resets the stack to a local area, and restores the CCP's stack before returning directly to the CCP. Thus, the dump program does not perform and warm start at the end of processing.

```
                ; DUMP program reads input file and displays hex data
                ;
0100                    org     100h
0005 =          bdos    equ     0005h   ;dos entry point
0001 =          cons    equ     1       ;read console
0002 =          typef   equ     2       ;type function
0009 =          printf  equ     9       ;buffer print entry
000b =          brkf    equ     11      ;break key function (true if char
000f =          openf   equ     15      ;file open
0014 =          readf   equ     20      ;read function
                ;
005c =          fcb     equ     5ch     ;file control block address
0080 =          buff    equ     80h     ;input disk buffer address
                ;
                ;       non graphic characters
000d =          cr      equ     0dh     ;carriage return
000a =          lf      equ     0ah     ;line feed
                ;
                ;       file control block definitions
005c =          fcbdn   equ     fcb+0   ;disk name
005d =          fcbfn   equ     fcb+1   ;file name
0065 =          fcbft   equ     fcb+9   ;disk file type (3 characters)
0068 =          fcbrl   equ     fcb+12  ;file's current reel number
006b =          fcbrc   equ     fcb+15  ;file's record count (0 to 128)
007c =          fcbcr   equ     fcb+32  ;current (next) record number (0
007d =          fcbln   equ     fcb+33  ;fcb length
                ;
                ;       set up stack
0100 210000             lxi     h,0
0103 39                 dad     sp
                ;       entry stack pointer in hl from the ccp
0104 221502             shld    oldsp
                ;       set sp to local stack area (restored at finis)
0107 315702             lxi     sp,stktop
                ;       read and print successive buffers
010a cdc101             call    setup   ;set up input file
010d feff               cpi     255     ;255 if file not present
010f c21b01             jnz     openok  ;skip if open is ok
                ;
                ;       file not there, give error message and return
0112 11f301             lxi     d,opnmsg
0115 cd9c01             call    err
0118 c35101             jmp     finis   ;to return
                ;
```

```
                  openok: ;open operation ok, set buffer index to end
011b 3e80                 mvi     a,80h
011d 321302               sta     ibp       ;set buffer pointer to 80h
                  ;       hl contains next address to print
0120 210000               lxi     h,0       ;start with 0000
                  ;
                  gloop:
0123 e5                   push    h         ;save line position
0124 cda201               call    gnb
0127 e1                   pop     h         ;recall line position
0128 da5101               jc      finis     ;carry set by gnb if end file
012b 47                   mov     b,a
                  ;       print hex values
                  ;       check for line fold
012c 7d                   mov     a,l
012d e60f                 ani     0fh       ;check low 4 bits
012f c24401               jnz     nonum
                  ;       print line number
0132 cd7201               call    crlf
                  ;
                  ;       check for break key
0135 cd5901               call    break
                  ;       accum lsb = 1 if character ready
0138 0f                   rrc               ;into carry
0139 da5101               jc      finis     ;don't print any more
                  ;
013c 7c                   mov     a,h
013d cd8f01               call    phex
0140 7d                   mov     a,l
0141 cd8f01               call    phex
                  nonum:
0144 23                   inx     h         ;to next line number
0145 3e20                 mvi     a,' '
0147 cd6501               call    pchar
014a 78                   mov     a,b
014b cd8f01               call    phex
014e c32301               jmp     gloop
                  ;
                  finis:
                  ;       end of dump, return to ccp
                  ;       (note that a jmp to 0000h reboots)
0151 cd7201               call    crlf
0154 2a1502               lhld    oldsp
0157 f9                   sphl
                  ;       stack pointer contains ccp's stack location
0158 c9                   ret               ;to the ccp
                  ;
                  ;
                  ;       subroutines
                  ;
                  break:  ;check break key (actually any key will do)
0159 e5d5c5               push h! push d! push b; environment saved
015c 0e0b                 mvi     c,brkf
015e cd0500               call    bdos
0161 c1d1e1               pop b! pop d! pop h; environment restored
```

```
0164 c9                ret
              ;
              pchar:   ;print a character
0165 e5d5c5            push h! push d! push b; saved
0168 0e02              mvi     c,typef
016a 5f                mov     e,a
016b cd0500            call    bdos
016e c1d1e1            pop b! pop d! pop h; restored
0171 c9                ret
              ;
              crlf:
0172 3e0d              mvi     a,cr
0174 cd6501            call    pchar
0177 3e0a              mvi     a,lf
0179 cd6501            call    pchar
017c c9                ret
              ;
              ;
              pnib:    ;print nibble in reg a
017d e60f              ani     0fh     ;low 4 bits
017f fe0a              cpi     10
0181 d28901            jnc     p10
              ;        less than or equal to 9
0184 c630              adi     '0'
0186 c38b01            jmp     prn
              ;
              ;        greater or equal to 10
0189 c637     p10:     adi     'a' - 10
018b cd6501   prn:     call    pchar
018e c9                ret
              ;
              phex:    ;print hex char in reg a
018f f5                push    psw
0190 0f                rrc
0191 0f                rrc
0192 0f                rrc
0193 0f                rrc
0194 cd7d01            call    pnib    ;print nibble
0197 f1                pop     psw
0198 cd7d01            call    pnib
019b c9                ret
              ;
              err:     ;print error message
              ;        d,e addresses message ending with "$"
019c 0e09              mvi     c,printf        ;print buffer function
019e cd0500            call    bdos
01a1 c9                ret
              ;
              ;
              gnb:     ;get next byte
01a2 3a1302            lda     ibp
01a5 fe80              cpi     80h
01a7 c2b301            jnz     g0
              ;        read another buffer
              ;
```

```
                ;
01aa cdce01           call    diskr
01ad b7               ora     a        ;zero value if read ok
01ae cab301           jz      g0       ;for another byte
                ;     end of data, return with carry set for eof
01b1 37               stc
01b2 c9               ret
                ;
                g0:   ;read the byte at buff+reg a
01b3 5f               mov     e,a      ;ls byte of buffer index
01b4 1600             mvi     d,0      ;double precision index to de
01b6 3c               inr     a        ;index=index+1
01b7 321302           sta     ibp      ;back to memory
                ;     pointer is incremented
                ;     save the current file address
01ba 218000           lxi     h,buff
01bd 19               dad     d
                ;     absolute character address is in hl
01be 7e               mov     a,m
                ;     byte is in the accumulator
01bf b7               ora     a        ;reset carry bit
01c0 c9               ret
                ;
                setup: ;set up file
                ;     open the file for input
01c1 af               xra     a        ;zero to accum
01c2 327c00           sta     fcbcr    ;clear current record
                ;
01c5 115c00           lxi     d,fcb
01c8 0e0f             mvi     c,openf
01ca cd0500           call    bdos
                ;     255 in accum if open error
01cd c9               ret
                ;
                diskr: ;read disk file record
01ce e5d5c5           push h! push d! push b
01d1 115c00           lxi     d,fcb
01d4 0e14             mvi     c,readf
01d6 cd0500           call    bdos
01d9 c1d1e1           pop b! pop d! pop h
01dc c9               ret
                ;
                ;     fixed message area
01dd 46494c0signon: db        'file dump version 2.0$'
01f3 0d0a4e0opnmsg: db        cr,lf,'no input file present on disk$'

                ;     variable area
0213            ibp:    ds    2        ;input buffer pointer
0215            oldsp:  ds    2        ;entry sp value from ccp
                ;
                ;     stack area
0217                    ds    64       ;reserve 32 level stack
                stktop:
                ;
0257                    end
```

5. A SAMPLE RANDOM ACCESS PROGRAM.

This manual is concluded with a rather extensive, but complete example of random access operation. The program listed below performs the simple function of reading or writing random records upon command from the terminal. Given that the program has been created, assembled, and placed into a file labelled RANDOM.COM, the CCP level command:

RANDOM X.DAT

starts the test program. The program looks for a file by the name X.DAT (in this particular case) and, if found, proceeds to prompt the console for input. If not found, the file is created before the prompt is given. Each prompt takes the form

next command?

and is followed by operator input, terminated by a carriage return. The input commands take the form

nW nR Q

where n is an integer value in the range 0 to 65535, and W, R, and Q are simple command characters corresponding to random write, random read, and quit processing, respectively. If the W command is issued, the RANDOM program issues the prompt

type data:

The operator then responds by typing up to 127 characters, followed by a carriage return. RANDOM then writes the character string into the X.DAT file at record n. If the R command is issued, RANDOM reads record number n and displays the string value at the console. If the Q command is issued, the X.DAT file is closed, and the program returns to the console command processor. In the interest of brevity, the only error message is

error, try again

The program begins with an initialization section where the input file is opened or created, followed by a continuous loop at the label "ready" where the individual commands are interpreted. The default file control block at 005CH and the default buffer at 0080H are used in all disk operations. The utility subroutines then follow, which contain the principal input line processor, called "readc." This particular program shows the elements of random access processing, and can be used as the basis for further program development.

```
                ;*****************************************************
                ;*                                                   *
                ;* sample random access program for cp/m 2.0         *
                ;*                                                   *
                ;*****************************************************
0100                    org     100h    ;base of tpa
                ;
0000 =          reboot  equ     0000h   ;system reboot
0005 =          bdos    equ     0005h   ;bdos entry point
                ;
0001 =          coninp  equ     1       ;console input function
0002 =          conout  equ     2       ;console output function
0009 =          pstring equ     9       ;print string until '$'
000a =          rstring equ     10      ;read console buffer
000c =          version equ     12      ;return version number
000f =          openf   equ     15      ;file open function
0010 =          closef  equ     16      ;close function
0016 =          makef   equ     22      ;make file function
0021 =          readr   equ     33      ;read random
0022 =          writer  equ     34      ;write random
                ;
005c =          fcb     equ     005ch   ;default file control block
007d =          ranrec  equ     fcb+33  ;random record position
007f =          ranovf  equ     fcb+35  ;high order (overflow) byte
0080 =          buff    equ     0080h   ;buffer address
                ;
000d =          cr      equ     0dh     ;carriage return
000a =          lf      equ     0ah     ;line feed
                ;
                ;*****************************************************
                ;*                                                   *
                ;* load SP, set-up file for random access            *
                ;*                                                   *
                ;*****************************************************
0100 31bc0              lxi     sp,stack
                ;
                ;       version 2.0?
0103 0e0c               mvi     c,version
0105 cd050              call    bdos
0108 fe20               cpi     20h     ;version 2.0 or better?
010a d2160              jnc     versok
                ;       bad version, message and go back
010d 111b0              lxi     d,badver
0110 cdda0              call    print
0113 c3000              jmp     reboot
                ;
                versok:
                ;       correct version for random access
0116 0e0f               mvi     c,openf ;open default fcb
0118 115c0              lxi     d,fcb
011b cd050              call    bdos
011e 3c                 inr     a       ;err 255 becomes zero
011f c2370              jnz     ready
                ;
                ;       cannot open file, so create it
```

```
0122 0e16           mvi     c,makef
0124 115c0          lxi     d,fcb
0127 cd050          call    bdos
012a 3c             inr     a       ;err 255 becomes zero
012b c2370          jnz     ready
                ;
                ;       cannot create file, directory full
012e 113a0          lxi     d,nospace
0131 cdda0          call    print
0134 c3000          jmp     reboot  ;back to ccp
                ;
                ;****************************************************
                ;*                                                  *
                ;*  loop back to "ready" after each command         *
                ;*                                                  *
                ;****************************************************
                ;
                ready:
                ;       file is ready for processing
                ;
0137 cde50          call    readcom ;read next command
013a 227d0          shld    ranrec  ;store input record#
013d 217f0          lxi     h,ranovf
0140 3600           mvi     m,0     ;clear high byte if set
0142 fe51           cpi     'Q'     ;quit?
0144 c2560          jnz     notq
                ;
                ;       quit processing, close file
0147 0e10           mvi     c,closef
0149 115c0          lxi     d,fcb
014c cd050          call    bdos
014f 3c             inr     a       ;err 255 becomes 0
0150 cab90          jz      error   ;error message, retry
0153 c3000          jmp     reboot  ;back to ccp
                ;
                ;****************************************************
                ;*                                                  *
                ;* end of quit command, process write               *
                ;*                                                  *
                ;****************************************************
                notq:
                ;       not the quit command, random write?
0156 fe57           cpi     'W'
0158 c2890          jnz     notw
                ;
                ;       this is a random write, fill buffer until cr
015b 114d0          lxi     d,datmsg
015e cdda0          call    print   ;data prompt
0161 0e7f           mvi     c,127   ;up to 127 characters
0163 21800          lxi     h,buff  ;destination
                rloop:  ;read next character to buff
0166 c5             push    b       ;save counter
0167 e5             push    h       ;next destination
0168 cdc20          call    getchr  ;character to a
016b e1             pop     h       ;restore counter
```

```
016c c1            pop     b       ;restore next to fill
016d fe0d          cpi     cr      ;end of line?
016f ca780         jz      erloop
          ;        not end, store character
0172 77            mov     m,a
0173 23            inx     h       ;next to fill
0174 0d            dcr     c       ;counter goes down
0175 c2660         jnz     rloop   ;end of buffer?
          erloop:
          ;        end of read loop, store 00
0178 3600          mvi     m,0
          ;
          ;        write the record to selected record number
017a 0e22          mvi     c,writer
017c 115c0         lxi     d,fcb
017f cd050         call    bdos
0182 b7            ora     a       ;error code zero?
0183 c2b90         jnz     error   ;message if not
0186 c3370         jmp     ready   ;for another record
          ;
          ;*******************************************************
          ;*                                                     *
          ;* end of write command, process read                  *
          ;*                                                     *
          ;*******************************************************
          notw:
          ;        not a write command, read record?
0189 fe52          cpi     'R'
018b c2b90         jnz     error   ;skip if not
          ;
          ;        read random record
018e 0e21          mvi     c,readr
0190 115c0         lxi     d,fcb
0193 cd050         call    bdos
0196 b7            ora     a       ;return code 00?
0197 c2b90         jnz     error
          ;
          ;        read was successful, write to console
019a cdcf0         call    crlf    ;new line
019d 0e80          mvi     c,128   ;max 128 characters
019f 21800         lxi     h,buff  ;next to get
          wloop:
01a2 7e            mov     a,m     ;next character
01a3 23            inx     h       ;next to get
01a4 e67f          ani     7fh     ;mask parity
01a6 ca370         jz      ready   ;for another command if 00
01a9 c5            push    b       ;save counter
01aa e5            push    h       ;save next to get
01ab fe20          cpi     ' '     ;graphic?
01ad d4c80         cnc     putchr  ;skip output if not
01b0 e1            pop     h
01b1 c1            pop     b
01b2 0d            dcr     c       ;count=count-1
01b3 c2a20         jnz     wloop
01b6 c3370         jmp     ready
```

```
            ;
            ;*****************************************************
            ;*                                                   *
            ;* end of read command, all errors end-up here       *
            ;*                                                   *
            ;*****************************************************
            ;
            error:
0lb9 11590          lxi     d,errmsg
0lbc cdda0          call    print
0lbf c3370          jmp     ready
            ;
            ;*****************************************************
            ;*                                                   *
            ;* utility subroutines for console i/o               *
            ;*                                                   *
            ;*****************************************************
            getchr:
                    ;read next console character to a
0lc2 0e01           mvi     c,coninp
0lc4 cd050          call    bdos
0lc7 c9             ret
            ;
            putchr:
                    ;write character from a to console
0lc8 0e02           mvi     c,conout
0lca 5f             mov     e,a     ;character to send
0lcb cd050          call    bdos    ;send character
0lce c9             ret
            ;
            crlf:
                    ;send carriage return line feed
0lcf 3e0d           mvi     a,cr    ;carriage return
0ld1 cdc80          call    putchr
0ld4 3e0a           mvi     a,lf    ;line feed
0ld6 cdc80          call    putchr
0ld9 c9             ret
            ;
            print:
                    ;print the buffer addressed by de until $
0lda d5             push    d
0ldb cdcf0          call    crlf
0lde d1             pop     d       ;new line
0ldf 0e09           mvi     c,pstring
0le1 cd050          call    bdos    ;print the string
0le4 c9             ret
            ;
            readcom:
                    ;read the next command line to the conbuf
0le5 116b0          lxi     d,prompt
0le8 cdda0          call    print   ;command?
0leb 0e0a           mvi     c,rstring
0led 117a0          lxi     d,conbuf
0lf0 cd050          call    bdos    ;read command line
            ;       command line is present, scan it
```

```
01f3 21000          lxi     h,0       ;start with 0000
01f6 117c0          lxi     d,conlin;command line
01f9 1a      readc: ldax    d         ;next command character
01fa 13             inx     d         ;to next command position
01fb b7             ora     a         ;cannot be end of command
01fc c8             rz
             ;      not zero, numeric?
01fd d630           sui     '0'
01ff fe0a           cpi     10        ;carry if numeric
0201 d2130          jnc     endrd
             ;      add-in next digit
0204 29             dad     h         ;*2
0205 4d             mov     c,l
0206 44             mov     b,h       ;bc = value * 2
0207 29             dad     h         ;*4
0208 29             dad     h         ;*8
0209 09             dad     b         ;*2 + *8 = *10
020a 85             add     l         ;+digit
020b 6f             mov     l,a
020c d2f90          jnc     readc     ;for another char
020f 24             inr     h         ;overflow
0210 c3f90          jmp     readc     ;for another char
             endrd:
             ;      end of read, restore value in a
0213 c630           adi     '0'       ;command
0215 fe61           cpi     'a'       ;translate case?
0217 d8             rc
             ;      lower case, mask lower case bits
0218 e65f           ani     101$1111b
021a c9             ret
             ;
             ;*****************************************************
             ;*                                                   *
             ;* string data area for console messages             *
             ;*                                                   *
             ;*****************************************************
             badver:
021b 536f79         db      'sorry, you need cp/m version 2$'
             nospace:
023a 4e6f29         db      'no directory space$'
             datmsg:
024d 547970         db      'type data: $'
             errmsg:
0259 457272         db      'error, try again.$'
             prompt:
026b 4e6570         db      'next command? $'
             ;
```

```
              ;*****************************************************
              ;*                                                   *
              ;* fixed and variable data area                      *
              ;*                                                   *
              ;*****************************************************
027a 21       conbuf: db      conlen  ;length of console buffer
027b          consiz: ds      1       ;resulting size after read
027c          conlin: ds      32      ;length 32 buffer
0021 =        conlen  equ     $-consiz
              ;
029c                  ds      32      ;16 level stack
              stack:
02bc                  end
```

Again, major improvements could be made to this particular program to enhance its operation. In fact, with some work, this program could evolve into a simple data base management system. One could, for example, assume a standard record size of 128 bytes, consisting of arbitrary fields within the record. A program, called GETKEY, could be developed which first reads a sequential file and extracts a specific field defined by the operator. For example, the command

GETKEY NAMES.DAT LASTNAME 10 20

would cause GETKEY to read the data base file NAMES.DAT and extract the "LASTNAME" field from each record, starting at position 10 and ending at character 20. GETKEY builds a table in memory consisting of each particular LASTNAME field, along with its 16-bit record number location within the file. The GETKEY program then sorts this list, and writes a new file, called LASTNAME.KEY, which is an alphabetical list of LASTNAME fields with their corresponding record numbers. (This list is called an "inverted index" in information retrieval parlance.).

Rename the program shown above as QUERY, and massage it a bit so that it reads a sorted key file into memory. The command line might appear as:

QUERY NAMES.DAT LASTNAME.KEY

Instead of reading a number, the QUERY program reads an alphanumeric string which is a particular key to find in the NAMES.DAT data base. Since the LASTNAME.KEY list is sorted, you can find a particular entry quite rapidly by performing a "binary search," similar to looking up a name in the telephone book. That is, starting at both ends of the list, you examine the entry halfway in between and, if not matched, split either the upper half or the lower half for the next search. You'll quickly reach the item you're looking for (in log2(n) steps) where you'll find the corresponding record number. Fetch and display this record at the console, just as we have done in the program shown above.

At this point you're just getting started. With a little more work, you can allow a fixed grouping size which differs from the 128 byte record shown above. This is accomplished by keeping track of the record number as well as the byte offset within the record. Knowing the group size, you randomly access the record containing the proper group, offset to the beginning of the group within the record read sequentially until the group size has been exhausted.

Finally, you can improve QUERY considerably by allowing boolean expressions which compute the set of records which satisfy several relationships, such as a LASTNAME between HARDY and LAUREL, and an AGE less than 45. Display all the records which fit this description. Finally, if your lists are getting too big to fit into memory, randomly access your key files from the disk as well. One note of consolation after all this work: if you make it through the project, you'll have no more need for this manual!

6. SYSTEM FUNCTION SUMMARY.

FUNC	FUNCTION NAME	INPUT PARAMETERS	OUTPUT RESULTS
0	System Reset	none	none
1	Console Input	none	A = char
2	Console Output	E = char	none
3	Reader Input	none	A = char
4	Punch Output	E = char	none
5	List Output	E = char	none
6	Direct Console I/O	see def	see def
7	Get I/O Byte	none	A = IOBYTE
8	Set I/O Byte	E = IOBYTE	none
9	Print String	DE = .Buffer	none
10	Read Console Buffer	DE = .Buffer	see def
11	Get Console Status	none	A = 00/FF
12	Return Version Number	none	HL= Version*
13	Reset Disk System	none	see def
14	Select Disk	E = Disk Number	see def
15	Open File	DE = .FCB	A = Dir Code
16	Close File	DE = .FCB	A = Dir Code
17	Search for First	DE = .FCB	A = Dir Code
18	Search for Next	none	A = Dir Code
19	Delete File	DE = .FCB	A = Dir Code
20	Read Sequential	DE = .FCB	A = Err Code
21	Write Sequential	DE = .FCB	A = Err Code
22	Make File	DE = .FCB	A = Dir Code
23	Rename File	DE = .FCB	A = Dir Code
24	Return Login Vector	none	HL= Login Vect*
25	Return Current Disk	none	A = Cur Disk#
26	Set DMA Address	DE = .DMA	none
27	Get Addr(Alloc)	none	HL= .Alloc
28	Write Protect Disk	none	see def
29	Get R/O Vector	none	HL= R/O Vect*
30	Set File Attributes	DE = .FCB	see def
31	Get Addr(disk parms)	none	HL= .DPB
32	Set/Get User Code	see def	see def
33	Read Random	DE = .FCB	A = Err Code
34	Write Random	DE = .FCB	A = Err Code
35	Compute File Size	DE = .FCB	r0, r1, r2
36	Set Random Record	DE = .FCB	r0, r1, r2

* Note that A = L, and B = H upon return

ED: A CONTEXT EDITOR FOR THE CP/M DISK SYSTEM

USER'S MANUAL

Table of Contents

ED USER'S MANUAL

1. ED TUTORIAL

1.1. Introduction to ED.

ED is the context editor for CP/M, and is used to create and alter CP/M source files. ED is initiated in CP/M by typing

$$\text{ED} \left\{ \begin{array}{l} \text{<filename>} \\ \\ \text{<filename>.<filetype>} \end{array} \right\}$$

In general, ED reads segments of the source file given by <filename> or <filename> . <filetype> into central memory, where the file is manipulated by the operator, and subsequently written back to disk after alterations. If the source file does not exist before editing, it is created by ED and initialized to empty. The overall operation of ED is shown in Figure 1.

1.2. ED Operation

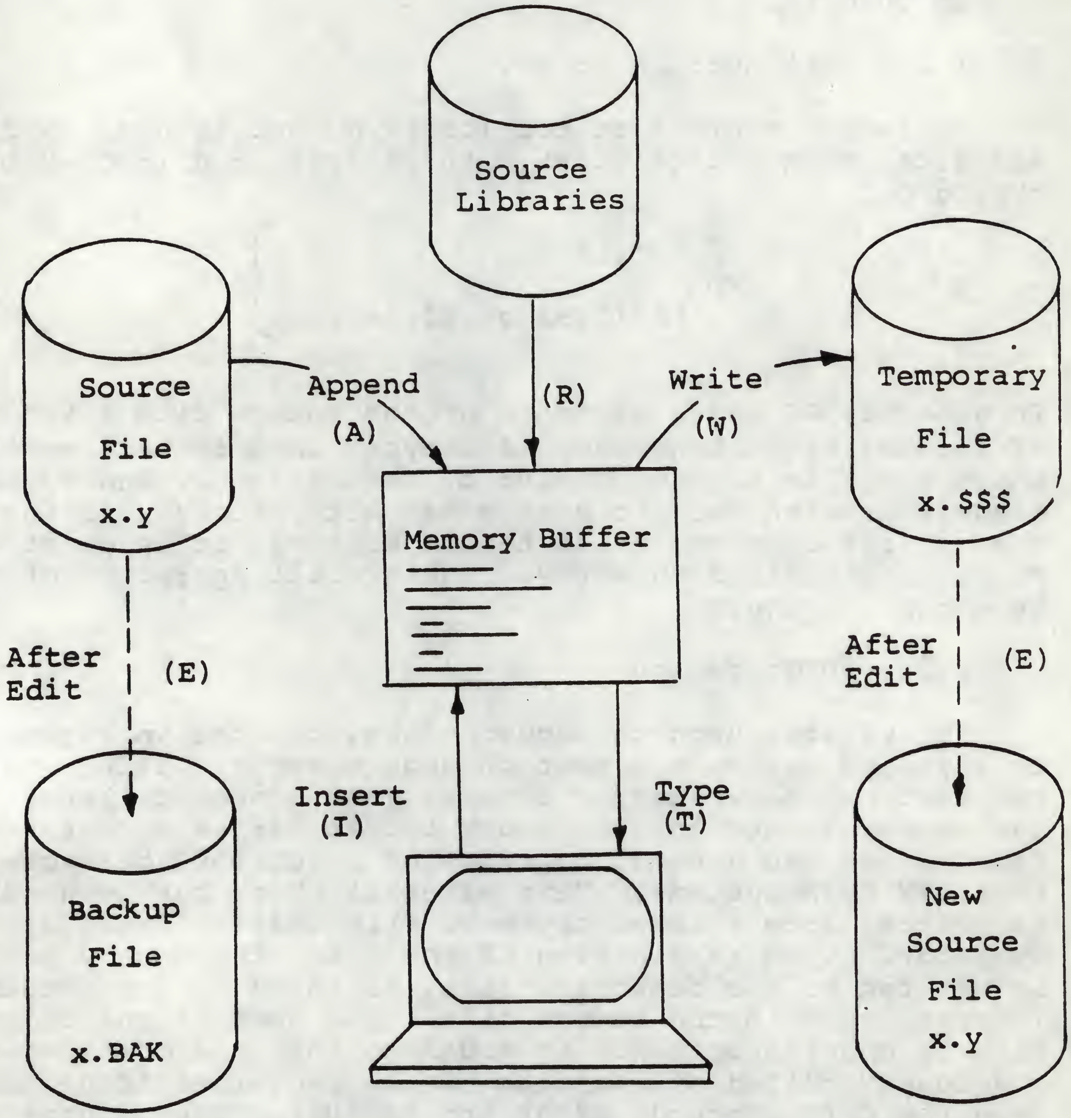
ED operates upon the source file, denoted in Figure 1 by x.y, and passes all text through a memory buffer where the text can be viewed or altered (the number of lines which can be maintained in the memory buffer varies with the line length, but has a total capacity of about 6000 characters in a 16K CP/M system). Text material which has been edited is written onto a temporary work file under command of the operator. Upon termination of the edit, the memory buffer is written to the temporary file, followed by any remaining (unread) text in the source file. The name of the original file is changed from x.y to x.BAK so that the most recent previously edited source file can be reclaimed if necessary (see the CP/M commands ERASE and RENAME). The temporary file is then changed from x.$$$ to x.y which becomes the resulting edited file.

The memory buffer is logically between the source file and working file as shown in Figure 2.

1.3. Text Transfer Functions

Given that n is an integer value in the range 0 through 65535, the following ED commands transfer lines of text from the source file through the memory buffer to the temporary (and eventually final) file:

Figure 1. Overall ED Operation



Note: the ED program accepts both lower and upper case ASCII characters as input from the console. Single letter commands can be typed in either case. The U command can be issued to cause ED to translate lower case alphabetics to upper case as characters are filled to the memory buffer from the console. Characters are echoed as typed without translation, however. The -U command causes ED to revert to "no translation" mode. ED starts with an assumed -U in effect.

Figure 2. Memory Buffer Organization

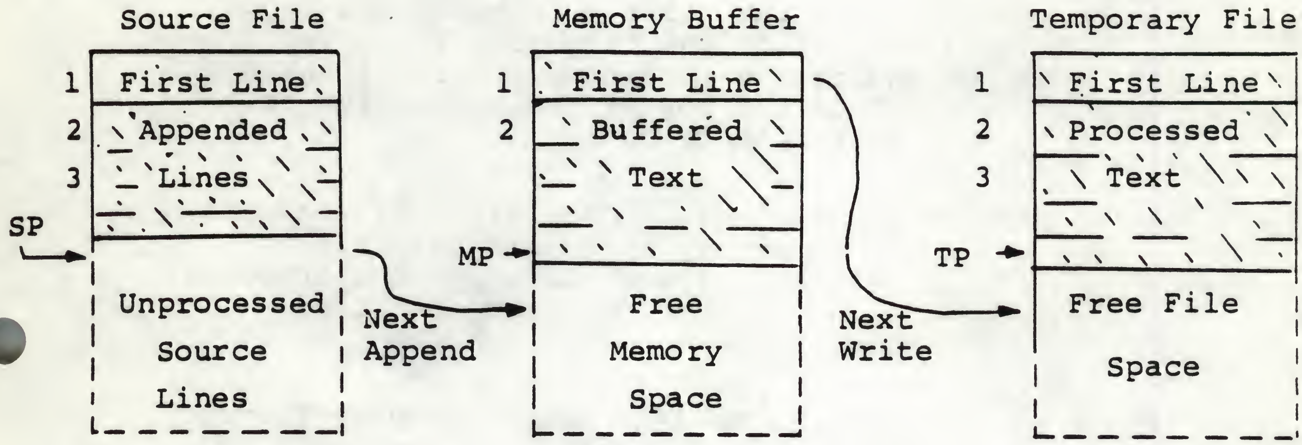
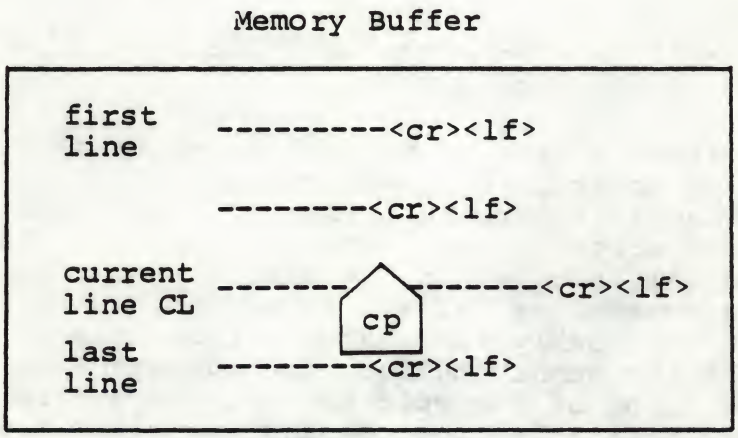


Figure 3. Logical Organization of Memory Buffer



nA<cr>* - append the next n unprocessed source lines from the source file at SP to the end of the memory buffer at MP. Increment SP and MP by n.

nW<cr> - write the first n lines of the memory buffer to the temporary file free space. Shift the remaining lines n+1 through MP to the top of the memory buffer. Increment TP by n.

E<cr> - end the edit. Copy all buffered text to temporary file, and copy all unprocessed source lines to the temporary file. Rename files as described previously.

H<cr> - move to head of new file by performing automatic E command. Temporary file becomes the new source file, the memory buffer is emptied, and a new temporary file is created (equivalent to issuing an E command, followed by a reinvocation of ED using x.y as the file to edit).

O<cr> - return to original file. The memory buffer is emptied, the temporary file id deleted, and the SP is returned to position 1 of the source file. The effects of the previous editing commands are thus nullified.

Q<cr> - quit edit with no file alterations, return to CP/M.

There are a number of special cases to consider. If the integer n is omitted in any ED command where an integer is allowed, then 1 is assumed. Thus, the commands A and W append one line and write 1 line, respectively. In addition, if a pound sign (#) is given in the place of n, then the integer 65535 is assumed (the largest value for n which is allowed). Since most reasonably sized source files can be contained entirely in the memory buffer, the command #A is often issued at the beginning of the edit to read the entire source file to memory. Similarly, the command #W writes the entire buffer to the temporary file. Two special forms of the A and W

---

*<cr> represents the carriage-return key

commands are provided as a convenience. The command 0A fills the current memory buffer to at least half-full, while 0W writes lines until the buffer is at least half empty. It should also be noted that an error is issued if the memory buffer size is exceded. The operator may then enter any command (such as W) which does not increase memory requirements. The remainder of any partial line read during the overflow will be brought into memory on the next successful append.

1.4. Memory Buffer Organization

The memory buffer can be considered a sequence of source lines brought in with the A command from a source file. The memory buffer has an associated (imaginary) character pointer CP which moves throughout the memory buffer under command of the operator. The memory buffer appears logically as shown in Figure 3 where the dashes represent characters of the source line of indefinite length, terminated by carriage-return (<cr>) and line-feed (<lf>) characters, and cp represents the imaginary character pointer. Note that the CP is always located ahead of the first character of the first line, behind the last character of the last line, or between two characters. The current line CL is the source line which contains the CP.

1.5. Memory Buffer Operation

Upon initiation of ED, the memory buffer is empty (ie, CP is both ahead and behind the first and last character). The operator may either append lines (A command) from the source file, or enter the lines directly from the console with the insert command

```
I<cr>
```

ED then accepts any number of input lines, where each line terminates with a <cr> (the <lf> is supplied automatically), until a control-z (denoted by ↑z is typed by the operator. The CP is positioned after the last character entered. The sequence

```
I<cr>
NOW IS THE<cr>
TIME FOR<cr>
ALL GOOD MEN<cr>
↑z
```

leaves the memory buffer as shown below

```
NOW IS THE<cr><lf>
TIME FOR<cr><lf>
ALL GOOD MEN<cr><lf>
                    cp
```

Various commands can then be issued which manipulate the CP or display source text in the vicinity of the CP. The commands shown below with a preceding n indicate that an optional unsigned value can be specified. When preceded by ±, the command can be unsigned, or have an optional preceding plus or minus sign. As before, the pound sign (#) is replaced by 65535. If an integer n is optional, but not supplied, then n=1 is assumed. Finally, if a plus sign is optional, but none is specified, then + is assumed.

±B<cr> - move CP to beginning of memory buffer if +, and to bottom if -.

±nC<cr> - move CP by ±n characters (toward front of buffer if +), counting the <cr><lf> as two distinct characters

±nD<cr> - delete n characters ahead of CP if plus and behind CP if minus.

±nK<cr> - kill (ie remove) ±n lines of source text using CP as the current reference. If CP is not at the beginning of the current line when K is issued, then the characters before CP remain if + is specified, while the characters after CP remain if - is given in the command.

±nL<cr> - if n=0 then move CP to the beginning of the current line (if it is not already there) if n≠0 then first move the CP to the beginning of the current line, and then move it to the beginning of the line which is n lines down (if +) or up (if -). The CP will stop at the top or bottom of the memory buffer if too large a value of n is specified.

±nT<cr> - If n=0 then type the contents of the current line up to CP. If n=1 then type the contents of the current line from CP to the end of the line. If n>1 then type the current line along with n-1 lines which follow, if + is specified. Similarly, if n>1 and - is given, type the previous n lines, up to the CP. The break key can be depressed to abort long type-outs.

±n<cr> - equivalent to ±nLT, which moves up or down and types a single line

1.6. Command Strings

Any number of commands can be typed contiguously (up to the capacity of the CP/M console buffer), and are executed only after the <cr> is typed. Thus, the operator may use the CP/M console command functions to manipulate the input command:

Rubout	remove the last character
Control-U	delete the entire line
Control-C	re-initialize the CP/M System
Control-E	return carriage for long lines without transmitting buffer (max 128 chars)

Suppose the memory buffer contains the characters shown in the previous section, with the CP following the last character of the buffer. The command strings shown below produce the results shown to the right

Command String	Effect	Resulting Memory Buffer
1. B2T<cr>	move to beginning of buffer and type 2 lines: "NOW IS THE TIME FOR"	cp NOW IS THE<cr><lf> TIME FOR<cr><lf> ALL GOOD MEN<cr><lf>
2. 5C0T<cr>	move CP 5 characters and type the beginning of the line "NOW I"	NOW I cp S THE<cr><lf>

3.	2L-T<cr>	move two lines down and type previous line "TIME FOR"	NOW IS THE<cr><lf> TIME FOR<cr><lf> cp ALL GOOD MEN<cr><lf>
4.	-L#K<cr>	move up one line, delte 65535 lines which follow	NOW IS THE<cr><lf> cp
5.	I<cr> TIME TO<cr> INSERT<cr> ↑z	insert two lines of text	NOW IS THE<cr><lf> TIME TO<cr><lf> INSERT<cr><lf> cp
6.	-2L#T<cr>	move up two lines, and type 65535 lines ahead of CP "NOW IS THE"	NOW IS THE<cr><lf> cp TIME TO<cr><lf> INSERT<cr><lf>
7.	<cr>	move down one line and type one line "INSERT"	NOW IS THE<cr><lf> TIME TO<cr><lf> cp INSERT<cr><lf>

1.7. Text Search and Alteration

ED also has a command which locates strings within the memory buffer. The command takes the form

$$nF\ c_1c_2\ldots c_k \left\{ \begin{matrix} \text{<cr>} \\ \uparrow z \end{matrix} \right\}$$

where c_1 through c_k represent the characters to match followed by either a <cr> or control -z*. ED starts at the current position of CP and attempts to match all k characters. The match is attempted n times, and if successful, the CP is moved directly after the character c_k. If the n matches are not successful, the CP is not moved from its initial position. Search strings can include ↑l (control-l), which is replaced by the pair of symbols <cr><lf>.

*The control-z is used if additional commands will be typed following the ↑z.

The following commands illustrate the use of the F command:

Command String	Effect	Resulting Memory Buffer
1. B#T<cr>	move to beginning and type entire buffer	NOW IS THE<cr><lf> TIME FOR<cr><lf> ALL GOOD MEN<cr><lf>
2. FS T<cr>	find the end of the string "S T"	NOW IS T HE<cr><lf>
3. FI↑z0TT	find the next "I" and type to the CP then type the remainder of the current line: "TIME FOR"	NOW IS THE<cr><lf> TI ME FOR<cr><lf> ALL GOOD MEN<cr><lf>

An abbreviated form of the insert command is also allowed, which is often used in conjunction with the F command to make simple textual changes. The form is:

I $c_1c_2\ldots c_n$↑z   or

I $c_1c_2\ldots c_n$<cr>

where c_1 through c_n are characters to insert. If the insertion string is terminated by a ↑z, the characters c_1 through c_n are inserted directly following the CP, and the CP is moved directly after character c_n. The action is the same if the command is followed by a <cr> except that a <cr><lf> is automatically inserted into the text following character c_n. Consider the following command sequences as examples of the F and I commands:

Command String	Effect	Resulting Memory Buffer
BITHIS IS ↑z<cr>	Insert "THIS IS " at the beginning of the text	THIS IS NOW THE <cr><lf> TIME FOR<cr><lf> ALL GOOD MEN<cr><lf>

FTIME↑z-4DIPLACE↑z<cr>	find "TIME" and delete it; then insert "PLACE"	THIS IS NOW THE<cr><lf> PLACE cp FOR<cr><lf> ALL GOOD MEN<cr><lf>
3FO↑z-3D5DICHANGES↑<cr>	find third occurrence of "O" (ie the second "O" in GOOD), delete previous 3 characters; then insert "CHANGES"	THIS IS NOW THE <cr><lf> PLACE FOR<cr><lf> ALL CHANGES cp <cr><lf>
-8CISOURCE<cr>	move back 8 characters and insert the line "SOURCE<cr><lf>"	THIS IS NOW THE<cr><lf> PLACE FOR<cr><lf> ALL SOURCE<cr><lf> cp CHANGES<cr><lf>

ED also provides a single command which combines the F and I commands to perform simple string substitutions. The command takes the form

$$n\ S\ c_1c_2\ldots c_k{\uparrow}z\ d_1d_2\ldots d_m \left\{\begin{matrix}\text{<cr>}\\ {\uparrow}z\end{matrix}\right\}$$

and has exactly the same effect as applying the command string

$$F\ c_1c_2\ldots c_k{\uparrow}z\text{-}k\mathrm{DI}d_1d_2\ldots d_m \left\{\begin{matrix}\text{<cr>}\\ {\uparrow}z\end{matrix}\right\}$$

a total of n times. That is, ED searches the memory buffer starting at the current position of CP and successively substitutes the second string for the first string until the end of buffer, or until the substitution has been performed n times.

As a convenience, a command similar to F is provided by ED which automatically appends and writes lines as the search proceeds. The form is

$$n\ N\ c_1c_2\ldots c_k \left\{\begin{matrix}\mathrm{cr}\\ {\uparrow}z\end{matrix}\right\}$$

which searches the entire source file for the nth occurrence of the string $c_1c_2\ldots c_k$ (recall that F fails if the string cannot be found in the current buffer). The operation of the

N command is precisely the same as F except in the case that the string cannot be found within the current memory buffer. In this case, the entire memory contents is written (ie, an automatic #W is issued). Input lines are then read until the buffer is at least half full, or the entire source file is exhausted. The search continues in this manner until the string has been found n times, or until the source file has been completely transferred to the temporary file.

A final line editing function, called the juxtaposition command takes the form

$$n\ J\ c_1c_2\ldots c_k\uparrow z\ d_1d_2\ldots d_m\uparrow z\ e_1e_2\ldots e_q \left\{ \begin{matrix} \text{<cr>} \\ \uparrow z \end{matrix} \right\}$$

with the following action applied n times to the memory buffer: search from the current CP for the next occurrence of the string $c_1c_2\ldots c_k$. If found, insert the string $d_1,d_2,\ldots,d_m$, and move CP to follow d_m. Then delete all characters following CP up to (but not including) the string $e_1,e_2,\ldots e_q$, leaving CP directly after d_m. If $e_1,e_2,\ldots e_q$ cannot be found, then no deletion is made. If the current line is

cp NOW IS THE TIME<cr><lf>

Then the command

JW ↑zWHAT↑z↑l<cr>

Results in

NOW WHAT cp <cr><lf>

(Recall that ↑l represents the pair <cr><lf> in search and substitute strings).

It should be noted that the number of characters allowed by ED in the F,S,N, and J commands is limited to 100 symbols.

1.8. Source Libraries

ED also allows the inclusion of source libraries during the editing process with the R command. The form of this command is

$$R\ f_1f_2..f_n{\uparrow}z \qquad \text{or}$$

$$R\ f_1f_2..f_n\text{<cr>}$$

where $f_1f_2..f_n$ is the name of a source file on the disk with as assumed filetype of 'LIB'. ED reads the specified file, and places the characters into the memory buffer after CP, in a manner similar to the I command. Thus, if the command

RMACRO<cr>

is issued by the operator, ED reads from the file MACRO.LIB until the end-of-file, and automatically inserts the characters into the memory buffer.

1.9. Repetitive Command Execution

The macro command M allows the ED user to group ED commands together for repeated evaluation. The M command takes the form:

$$n\ M\ c_1c_2...c_k \begin{Bmatrix}\text{<cr>}\\ {\uparrow}z\end{Bmatrix}$$

where $c_1c_2...c_k$ represent a string of ED commands, not including another M command. ED executes the command string n times if n>1. If n=0 or 1, the command string is executed repetitively until an error condition is encountered (e.g., the end of the memory buffer is reached with an F command).

As an example, the following macro changes all occurrences of GAMMA to DELTA within the current buffer, and types each line which is changed:

MFGAMMA↑z-5DIDELTA↑z0TT<cr>

or equivalently

MSGAMMA↑zDELTA↑z0TT<cr>

2. ED ERROR CONDITIONS

On error conditions, ED prints the last character read before the error, along with an error indicator:

?	unrecognized command
>	memory buffer full (use one of the commands D,K,N,S, or W to remove characters), F,N, or S strings too long.
#	cannot apply command the number of times specified (e.g., in F command)
O	cannot open LIB file in R command

Cyclic redundancy check (CRC) information is written with each output record under CP/M in order to detect errors on subsequent read operations. If a CRC error is detected, CP/M will type

PERM ERR DISK d

where d is the currently selected drive (A,B,...). The operator can choose to ignore the error by typing any character at the console (in this case, the memory buffer data should be examined to see if it was incorrectly read), or the user can reset the system and reclaim the backup file, if it exists. The file can be reclaimed by first typing the contents of the BAK file to ensure that it contains the proper information:

TYPE x.BAK<cr>

where x is the file being edited. Then remove the primary file:

ERA x.y<cr>

and rename the BAK file:

REN x.y=x.BAK<cr>

The file can then be re-edited, starting with the previous version.

3. CONTROL CHARACTERS AND COMMANDS

The following table summarizes the control characters and commands available in ED:

Control Character	Function
↑c	system reboot
↑e	physical <cr><lf> (not actually entered in command)
↑i	logical tab (cols 1,8, 15,...)
↑l	logical <cr><lf> in search and substitute strings
↑u	line delete
↑z	string terminator
rubout	character delete
break	discontinue command (e.g., stop typing)

Command	Function
nA	append lines
±B	begin bottom of buffer
±nC	move character positions
±nD	delete characters
E	end edit and close files (normal end)
nF	find string
H	end edit, close and reopen files
I	insert characters
nJ	place strings in juxtaposition
±nK	kill lines
±nL	move down/up lines
nM	macro definition
nN	find next occurrence with autoscan
O	return to original file
±nP	move and print pages
Q	quit with no file changes
R	read library file
nS	substitute strings
±nT	type lines
± U	translate lower to upper case if U, no translation if -U
nW	write lines
nZ	sleep
±n<cr>	move and type (±nLT)

Appendix A: ED 1.4 Enhancements

The ED context editor contains a number of commands which enhance its usefulness in text editing. The improvements are found in the addition of line numbers, free space interrogation, and improved error reporting.

The context editor issued with CP/M 1.4 produces absolute line number prefixes when the "V" (Verify Line Numbers) command is issued. Following the V command, the line number is displayed ahead of each line in the format:

nnnnn:

where nnnnn is an absolute line number in the range 1 to 65535. If the memory buffer is empty, or if the current line is at the end of the memory buffer, then nnnnn appears as 5 blanks.

The user may reference an absolute line number by preceding any command by a number followed by a colon, in the same format as the line number display. In this case, the ED program moves the current line reference to the absolute line number, if the line exists in the current memory buffer. Thus, the command

345:T

is interpreted as "move to absolute line 345, and type the line." Note that absolute line numbers are produced only during the editing process, and are not recorded with the file. In particular, the line numbers will change following a deleted or expanded section of text.

The user may also reference an absolute line number as a backward or forward distance from the current line by preceding the absolute line number by a colon. Thus, the command

:400T

is interpreted as "type from the current line number through the line whose absolute number is 400." Combining the two line reference forms, the command

345::400T

for example, is interpreted as "move to absolute line 345, then type through absolute line 400." Note that absolute line references of this sort can precede any of the standard ED commands.

A special case of the V command, "0V", prints the memory buffer statistics in the form:

free/total

where "free" is the number of free bytes in the memory buffer (in decimal), and "total" is the size of the memory buffer.

ED 1.4 also includes a "block move" facility implemented through the "X" (Xfer) command. The form

nX

transfers the next n lines from the current line to a temporary file called

X$$$$$$.LIB

which is active only during the editing process. In general, the user can reposition the current line reference to any portion of the source file and transfer lines to the temporary file. The transferred line accumulate one after another in this file, and can be retrieved by simply typing:

R

which is the trivial case of the library read command. In this case, the entire transferred set of lines is read into the memory buffer. Note that the X command does not remove the transferred lines from the memory buffer, although a K command can be used directly after the X, and the R command does not empty the transferred line file. That is, given that a set of lines has been transferred with the X command, they can be re-read any number of times back into the source file. The command

ØX

is provided, however, to empty the transferred line file.

Note that upon normal completion of the ED program through Q or E, the temporary LIB file is removed. If ED is aborted through ctl-C, the LIB file will exist if lines have been transferred, but will generally be empty (a subsequent ED invocation will erase the temporary file).

Due to common typographical errors, ED 1.4 requires several potentially disasterous commands to be typed as single letters, rather than in composite commands. The commands

E (end), H (head), O (original), Q (quit)

must be typed as single letter commands.

ED 1.4 also prints error messages in the form

BREAK "x" AT c

where x is the error character, and c is the command where the error occurred.

CP/M DYNAMIC DEBUGGING TOOL (DDT)

USER'S GUIDE

Table of Contents

CP/M Dynamic Debugging Tool (DDT)

User's Guide

I. Introduction

The DDT program allows dynamic interactive testing and debugging of programs generated in the CP/M environment. The debugger is initiated by typing one of the following commands at the CP/M Console Command level:

```
DDT
DDT filename.HEX
DDT filename.COM
```

where "filename" is the name of the program to be loaded and tested. In both cases, the DDT program is brought into main memory in the place of the Console Command Processor (refer to the CP/M Interface Guide for standard memory organization), and thus resides directly below the Basic Disk Operating System portion of CP/M. The BDOS starting address, which is located in the address field of the JMP instruction at location 5H, is altered to reflect the reduced Transient Program Area size.

The second and third forms of the DDT command shown above perform the same actions as the first, except there is a subsequent automatic load of the specified HEX or COM file. The action is identical to the sequence of commands

```
DDT
Ifilename.HEX or Ifilename.COM
R
```

where the I and R commands set up and read the specified program to test (see the explanation of the I and R commands below for exact details).

Upon initiation, DDT prints the following sign-on message.

```
DDT VERS n.n
```

where nn represents the version number.

Following the sign on message, DDT prompts the operator with the character "-" and waits for input commands from the console. The operator can type any of several single character commands, terminated by a carriage return to execute the command. Each line of input can be line-edited using the standard CP/M controls

rubout	remove the last character typed
ctl-U	remove the entire line, ready for re-typing
ctl-C	system reboot

Any command can be up to 32 characters in length (an automatic carriage return is inserted as the 33rd character), where the first character determines the command type

A	enter assembly language mnemonics with operands
D	display memory in hexadecimal and ASCII
F	fill memory with constant data
G	begin execution with optional breakpoints
I	set up a standard input file control block
L	list memory using assembler mnemonics
M	move a memory segment from source to destination
R	read program for subsequent testing
S	substitute memory values
T	trace program execution
U	untraced program monitoring
X	examine and optionally alter the CPU state

The command character, in some cases, is followed by zero, one, two, or three hexadecimal values which are separated by commas or single blank characters. All DDT numeric output is in hexadecimal form. In all cases, the commands are not executed until the carriage return is typed at the end of the command.

At any point in the debug run, the operator can stop execution of DDT using either a ctl-C or GØ (jmp to location ØØØØH), and save the current memory image using a SAVE command of the form

```
SAVE n filename.COM
```

where n is the number of pages (256 byte blocks) to be saved on disk. The number of blocks can be determined by taking the high order byte of the top load address and converting this number to decimal. For example, if the highest address in the Transient Program Area is 1234H then the number of pages is 12H, or 18 in decimal. Thus the operator could type a ctl-C during the debug run, returning to the Console Processor level, followed by

```
SAVE 18 X.COM
```

The memory image is saved as X.COM on the diskette, and can be directly executed by simply typing the name X. If further testing is required, the memory image can be recalled by typing

DDT X.COM

which reloads previously saved program from loaction 100H through page 18 (12FFH). The machine state is not a part of the COM file, and thus the program must be restarted from the beginning in order to properly test it.

II. DDT COMMANDS.

The individual commands are given below in some detail. In each case, the operator must wait for the prompt character (-) before entering the command. If control is passed to a program under test, and the program has not reached a breakpoint, control can be returned to DDT by executing a RST 7 from the front panel (note that the rubout key should be used instead if the program is executing a T or U command). In the explanation of each command, the command letter is shown in some cases with numbers separated by commas, where the numbers are represented by lower case letters. These numbers are always assumed to be in a hexadecimal radix, and from one to four digits in length (longer numbers will be automatically truncated on the right).

Many of the commands operate upon a "CPU state" which corresponds to the program under test. The CPU state holds the registers of the program being debugged, and initially contains zeroes for all registers and flags except for the program counter (P) and stack pointer (S), which default to 100H. The program counter is subsequently set to the starting address given in the last record of a HEX file if a file of this form is loaded (see the I and R commands).

1. The A (Assemble) Command. DDT allows inline assembly language to be inserted into the current memory image using the A command which takes the form

As

where s is the hexadecimal starting address for the inline assembly. DDT prompts the console with the address of the next instruction to fill, and reads the console, looking for assembly language mnemonics (see the Intel 8080 Assembly Language Reference Card for a list of mnemonics), followed by register references and operands in absolute hexadecimal form. Each sucessive load address is printed before reading the console. The A command terminates when the first empty line is input from the console.

Upon completion of assembly language input, the operator can review the memory segment using the DDT disassembler (see the L command).

Note that the assembler/disassembler portion of DDT can be overlayed by the transient program being tested, in which case the DDT program responds with an error condition when the A and L commands are used (refer to Section IV).

2. The D (Display) Command. The D command allows the operator to view the contents of memory in hexadecimal and ASCII formats. The forms are

```
D
Ds
Ds,f
```

In the first case, memory is displayed from the current display address (initially 100H), and continues for 16 display lines. Each display line takes the form shown below

```
aaaa bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb cccccccccccccccc
```

where aaaa is the display address in hexadecimal, and bb represents data present in memory starting at aaaa. The ASCII characters starting at aaaa are given to the right (represented by the sequence of c's), where non-graphic characters are printed as a period (.) symbol. Note that both upper and lower case alphabetics are displayed, and thus will appear as upper case symbols on a console device that supports only upper case. Each display line gives the values of 16 bytes of data, except that the first line displayed is truncated so that the next line begins at an address which is a multiple of 16.

The second form of the D command shown above is similar to the first, except that the display address is first set to address s. The third form causes the display to continue from address s through address f. In all cases, the display address is set to the first address not displayed in this command, so that a continuing display can be accomplished by issuing successive D commands with no explicit addresses.

Excessively long displays can be aborted by pushing the rubout key.

3. The F (Fill) Command. The F command takes the form

```
Fs,f,c
```

where s is the starting address, f is the final address, and c is a hexadecimal byte constant. The effect is as follows: DDT stores the constant c at address s, increments the value of s and tests against f. If s exceeds f then the operation terminates, otherwise the operation is repeated. Thus, the fill command can be used to set a memory block to a specific constant value.

4. The G (Go) Command. Program execution is started using the G comand, with up to two optional breakpoint addresses. The G command takes one ot the forms

```
G
Gs
Gs,b
```

```
Gs,b,c
G,b
G,b,c
```

The first form starts execution of the program under test at the current value of the program counter in the current machine state, with no breakpoints set (the only way to regain control in DDT is through a RST 7 execution). The current program counter can be viewed by typing an X or XP command. The second form is similar to the first except that the program counter in the current machine state is set to address s before execution begins. The third form is the same as the second, except that program execution stops when address b is encountered (b must be in the area of the program under test). The instruction at location b is not executed when the breakpoint is encountered. The fourth form is identical to the third, except that two breakpoints are specified, one at b and the other at c. Encountering either breakpoint causes execution to stop, and both breakpoints are subsequently cleared. The last two forms take the program counter from the current machine state, and set one and two breakpoints, respectively.

Execution continues from the starting address in real-time to the next breakpoint. That is, there is no intervention between the starting address and the break address by DDT. Thus, if the program under test does not reach a breakpoint, control cannot return to DDT without executing a RST 7 instruction. Upon encountering a breakpoint, DDT stops execution and types

```
*d
```

where d is the stop address. The machine state can be examined at this point using the X (Examine) command. The operator must specify breakpoints which differ from the program counter address at the beginning of the G command. Thus, if the current program counter is 1234H, then the commands

```
G,1234
```

and

```
G400,400
```

both produce an immediate breakpoint, without executing any instructions whatsoever.

5. The I (Input) Command. The I command allows the operator to insert a file name into the default file control block at 5CH (the file control block created by CP/M for transient programs is placed at this location; see the CP/M Interface Guide). The default FCB can be used by the program under test as if it had been passed by the CP/M Console Processor. Note that this file name is also used by DDT for reading additional HEX and COM files. The form of the I command is

```
Ifilename
```

or

Ifilename.filetype

If the second form is used, and the filetype is either HEX or COM, then subsequent R commands can be used to read the pure binary or hex format machine code (see the R command for further details).

6. The L (List) Command. The L command is used to list assembly language mnemonics in a particular program region. The forms are

```
L
Ls
Ls,f
```

The first command lists twelve lines of disassembled machine code from the current list address. The second form sets the list address to s, and then lists twelve lines of code. The last form lists disassembled code from s through address f. In all three cases, the list address is set to the next unlisted location in preparation for a subsequent L command. Upon encountering an execution breakpoint, the list address is set to the current value of the program counter (see the G and T commands). Again, long typeouts can be aborted using the rubout key during the list process.

7. The M (Move) Command. The M command allows block movement of program or data areas from one location to another in memory. The form is

```
Ms,f,d
```

where s is the start address of the move, f is the final address of the move, and d is the destination address. Data is first moved from s to d, and both addresses are incremented. If s exceeds f then the move operation stops, otherwise the move operation is repeated.

8. The R (Read) Command. The R command is used in conjunction with the I command to read COM and HEX files from the diskette into the transient program area in preparation for the debug run. The forms are

```
R
Rb
```

where b is an optional bias address which is added to each program or data address as it is loaded. The load operation must not overwrite any of the system parameters from 000H through 0FFH (i.e., the first page of memory). If b is omitted, then b=0000 is assumed. The R command requires a previous I command, specifying the name of a HEX or COM file. The load address for each record is obtained from each individual HEX record, while an assumed load address of 100H is taken for COM files. Note that any number of R commands can be issued following the I command to re-read the program under test,

assuming the tested program does not destroy the default area at 5CH. Further, any file specified with the filetype "COM" is assumed to contain machine code in pure binary form (created with the LOAD or SAVE command), and all others are assumed to contain machine code in Intel hex format (produced, for example, with the ASM command).

Recall that the command

```
DDT filename.filetype
```

which initiates the DDT program is equivalent to the commands

```
DDT
-Ifilename.filetype
-R
```

Whenever the R command is issued, DDT responds with either the error indicator "?" (file cannot be opened, or a checksum error occurred in a HEX file), or with a load message taking the form

```
NEXT  PC
nnnn pppp
```

where nnnn is the next address following the loaded program, and pppp is the assumed program counter (1ØØH for COM files, or taken from the last record if a HEX file is specified).

9. The S (Set) Command. The S command allows memory locations to be examined and optionally altered. The form of the command is

```
Ss
```

where s is the hexadecimal starting address for examination and alteration of memory. DDT responds with a numeric prompt, giving the memory location, along with the data currently held in the memory location. If the operator types a carriage return, then the data is not altered. If a byte value is typed, then the value is stored at the prompted address. In either case, DDT continues to prompt with successive addresses and values until either a period (.) is typed by the operator, or an invalid input value is detected.

1Ø. The T (Trace) Command. The T command allows selective tracing of program execution for 1 to 65535 program steps. The forms are

```
T
Tn
```

In the first case, the CPU state is displayed, and the next program step is executed. The program terminates immediately, with the termination address

displayed as

    *hhhh

where hhhh is the next address to execute. The display address (used in the D command) is set to the value of H and L, and the list address (used in the L command) is set to hhhh. The CPU state at program termination can then be examined using the X command.

The second form of the T command is similar to the first, except that execution is traced for n steps (n is a hexadecimal value) before a program breakpoint is occurs. A breakpoint can be forced in the trace mode by typing a rubout character. The CPU state is displayed before each program step is taken in trace mode. The format of the display is the same as described in the X command.

Note that program tracing is discontinued at the interface to CP/M, and resumes after return from CP/M to the program under test. Thus, CP/M functions which access I/O devices, such as the diskette drive, run in real-time, avoiding I/O timing problems. Programs running in trace mode execute approximately 500 times slower than real time since DDT gets control after each user instruction is executed. Interrupt processing routines can be traced, but it must be noted that commands which use the breakpoint facility (G, T, and U) accomplish the break using a RST 7 instruction, which means that the tested program cannot use this interrupt location. Further, the trace mode always runs the tested program with interrupts enabled, which may cause problems if asynchronous interrupts are received during tracing.

Note also that the operator should use the rubout key to get control back to DDT during trace, rather than executing a RST 7, in order to ensure that the trace for the current instruction is completed before interruption.

11. The U (Untrace) Command. The U command is identical to the T command except that intermediate program steps are not displayed. The untrace mode allows from 1 to 65535 (0FFFFH) steps to be executed in monitored mode, and is used principally to retain control of an executing program while it reaches steady state conditions. All conditions of the T command apply to the U command.

12. The X (Examine) Command. The X command allows selective display and alteration of the current CPU state for the program under test. The forms are

    X
    Xr

where r is one of the 8080 CPU registers

    C   Carry Flag   (0/1)
    Z   Zero Flag   (0/1)

M	Minus Flag	(0/1)
E	Even Parity Flag	(0/1)
I	Interdigit Carry	(0/1)
A	Accumulator	(0-FF)
B	BC register pair	(0-FFFF)
D	DE register pair	(0-FFFF)
H	HL register pair	(0-FFFF)
S	Stack Pointer	(0-FFFF)
P	Program Counter	(0-FFFF)

In the first case, the CPU register state is displayed in the format

```
CfZfMfEfIf A=bb B=dddd D=dddd H=dddd S=dddd P=dddd inst
```

where f is a 0 or 1 flag value, bb is a byte value, and dddd is a double byte quantity corresponding to the register pair. The "inst" field contains the disassembled instruction which occurs at the location addressed by the CPU state's program counter.

The second form allows display and optional alteration of register values, where r is one of the registers given above (C, Z, M, E, I, A, B, D, H, S, or P). In each case, the flag or register value is first displayed at the console. The DDT program then accepts input from the console. If a carriage return is typed, then the flag or register value is not altered. If a value in the proper range is typed, then the flag or register value is altered. Note that BC, DE, and HL are displayed as register pairs. Thus, the operator types the entire register pair when B, C, or the BC pair is altered.

III. IMPLEMENTATION NOTES.

The organization of DDT allows certain non-essential portions to be overlayed in order to gain a larger transient program area for debugging large programs. The DDT program consists of two parts: the DDT nucleus and the assembler/disassembler module. The DDT nucleus is loaded over the Console Command Processor, and, although loaded with the DDT nucleus, the assembler/disassembler is overlayable unless used to assemble or disassemble.

In particular, the BDOS address at location 6H (address field of the JMP instruction at location 5H) is modified by DDT to address the base location of the DDT nucleus which, in turn, contains a JMP instruction to the BDOS. Thus, programs which use this address field to size memory see the logical end of memory at the base of the DDT nucleus rather than the base of the BDOS.

The assembler/disassembler module resides directly below the DDT nucleus in the transient program area. If the A, L, T, or X commands are used during the debugging process then the DDT program again alters the address field at 6H to include this module, thus further reducing the logical end of memory. If a program loads beyond the beginning of the assembler/disassembler module, the A and L commands are lost (their use produces a "?" in response), and the

trace and display (T and X) commands list the "inst" field of the display in hexadecimal, rather than as a decoded instruction.

IV. AN EXAMPLE.

The following example shows an edit, assemble, and debug for a simple program which reads a set of data values and determines the largest value in the set. The largest value is taken from the vector, and stored into "LARGE" at the termination of the program

```
ED SCAN.ASM
                                  tab character    rubout   rubout echo
*I
    ↑-I  ORG  ↑-I   100H      L L;START OF TRANSIENT AREA
         MVI        B,LEN     ;LENGTH OF VECTOR TO SCAN
         MVI        C,0       ;LARGER RST VALUE SO FAR
LOOP__P_O_O_L       LXI       H,VECT   ;BASE OF VECTOR
LOOP:    MOV        A,M       ;GET VALUE
         SUB        C         ;LARGER VALUE IN C?
  Rubout JNC        NFOUND    ;JUMP IF LARGER VALUE NOT FOUND
; deletes character
         NEW LARGEST VALUE, STORE IT TO C
         MOV        C,A
NFOUND:  INX        H         ;TO NEXT ELEMENT
         DCR        B         ;MORE TO SCAN?
         JNZ        LOOP      ;FOR ANOTHER
;
;        END OF SCAN, STORE C
         MOV        A,C       ;GET LARGEST VALUE
         STA        LARGE
         JMP        0         ;REBOOT
;
;        TEST DATA
VECT:    DB         2,0,4,3,5,6,1,5
LEN      EQU        $-VECT    ;LENGTH
LARGE:   DS         1         ;LARGEST VALUE ON EXIT
         END
↑z *B0P
         ORG        100H      ;START OF TRANSIENT AREA
         MVI        B,LEN     ;LENGTH OF VECTOR TO SCAN
         MVI        C,0       ;LARGEST VALUE SO FAR
         LXI        H,VECT    ;BASE OF VECTOR
LOOP:    MOV        A,M       ;GET VALUE
         SUB        C         ;LARGER VALUE IN C?
         JNC        NFOUND    ;JUMP IF LARGER VALUE NOT FOUND
;        NEW LARGEST VALUE, STORE IT TO C
         MOV        C,A
NFOUND:  INX        H         ;TO NEXT ELEMENT
         DCR        B         ;MORE TO SCAN?
         JNZ        LOOP      ;FOR ANOTHER
```

Create Source Program - underlined characters typed by programmer.
"↲" represents carriage return.

```
;       END OF SCAN, STORE C
        MOV     A,C     ;GET LARGEST VALUE
        STA     LARGE
        JMP     0       ;REBOOT
;
;       TEST DATA
VECT:   DB      2,0,4,3,5,6,1,5
LEN     EQU     $-VECT  ;LENGTH
LARGE:  DS      1       ;LARGEST VALUE ON EXIT
        END
*E
```

← End of Edit

```
ASM SCAN
```

Start Assembler

```
CP/M ASSEMBLER - VER 1.0

0122
002H USE FACTOR
END OF ASSEMBLY
```

Assembly Complete - Look at Program Listing

```
TYPE SCAN.PRN
```

Code Address — Machine Code — Source Program

```
 0100                    ORG     100H     ;START OF TRANSIENT AREA
 0100 0608               MVI     B,LEN    ;LENGTH OF VECTOR TO SCAN
 0102 0E00               MVI     C,0      ;LARGEST VALUE SO FAR
 0104 211901             LXI     H,VECT   ;BASE OF VECTOR
 0107 7E         LOOP:   MOV     A,M      ;GET VALUE
 0108 91                 SUB     C        ;LARGER VALUE IN C?
 0109 D20D01             JNC     NFOUND   ;JUMP IF LARGER VALUE NOT FOUND
                 ;       NEW LARGEST VALUE, STORE IT TO C
 010C 4F                 MOV     C,A
 010D 23         NFOUND: INX     H        ;TO NEXT ELEMENT
 010E 05                 DCR     B        ;MORE TO SCAN?
 010F C20701             JNZ     LOOP     ;FOR ANOTHER
                 ;
                 ;       END OF SCAN, STORE C
 0112 79                 MOV     A,C      ;GET LARGEST VALUE
 0113 322101             STA     LARGE
 0116 C30000             JMP     0        ;REBOOT
                 ;
                 ;       TEST DATA
 0119 0200040305VECT:    DB      2,0,4,3,5,6,1,5
 0008 =          LEN     EQU     $-VECT   ;LENGTH
 0121            LARGE:  DS      1        ;LARGEST VALUE ON EXIT
 0122                    END
A>
```

Code/data listing truncated

Value of Equate

```
DDT SCAN.HEX↲          Start Debugger using hex format machine code
16K DDT VER 1.0
NEXT  PC
0121 0000
-X↲                    last load address +1
                                                                  next instruction
                                                                  to execute at
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0000 OUT  7F        PC=0
-XP↲                   Examine registers before debug run
P=0000 100↲  Change PC to 100
-X↲ Look at registers again                               PC changed.
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI  B,08
-L100↲                                                    Next instruction
                                                          to execute at PC=100
0100  MVI  B,08
0102  MVI  C,00
0104  LXI  H,0119
0107  MOV  A,M
0108  SUB  C
0109  JNC  010D      Disassembled Machine
010C  MOV  C,A       Code at 100H.
010D  INX  H         (See Source Listing
010E  DCR  B         for comparison)
010F  JNZ  0107
0112  MOV  A,C
-L↲
0113  STA  0121
0116  JMP  0000
0119  STAX B
011A  NOP
011B  INR  B         A little more
011C  INX  B         machine code
011D  DCR  B         (note that Program
011E  MVI  B,01      ends at location 116
0120  DCR  B         with a JMP to 0000)
0121  LXI  D,2200
0124  LXI  H,0200
-A116↲ enter inline assembly mode to change the JMP to 0000 into a RST 7, which
       will cause the program under test to return to DDT if 116H
0116  RST 7↲   is ever executed.
0117↲ (single carriage return stops assemble mode)
-L113↲ List Code at 113H to check that RST 7 was properly inserted
0113  STA  0121    In place of JMP
0116  RST  07
```

```
0117   NOP
0118   NOP
0119   STAX B
011A   NOP
011B   INR  B
011C   INX  B
-
```

-X Look at registers

```
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI  B,08
```

-I Execute Program for one step. initial CPU state, before is executed

```
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 MVI  B,08*0102
```

-I Trace one step again (note 08H in B) automatic breakpoint

```
C0Z0M0E0I0 A=00 B=0800 D=0000 H=0000 S=0100 P=0102 MVI  C,00*0104
```

-I Trace again (Register C is cleared)

```
C0Z0M0E0I0 A=00 B=0800 D=0000 H=0000 S=0100 P=0104 LXI  H,0119*0107
```

-T3 Trace three steps

```
C0Z0M0E0I0 A=00 B=0800 D=0000 H=0119 S=0100 P=0107 MOV  A,M
C0Z0M0E0I0 A=02 B=0800 D=0000 H=0119 S=0100 P=0108 SUB  C
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0109 JNC  010D*010D
```

-D119 Display memory starting at 119H. automatic breakpoint at 10DH

Program data

Lowercase x

```
0119 02 00 04 03 05 06 01                            .......
0120 05 11 00 22 21 00 02 7E EB 77 13 23 EB 0B 78 B1 ..."!..^.W.#..x.
0130 C2 27 01 C3 03 29 00 00 00 00 00 00 00 00 00 00 .'...)..........
0140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
01A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
01B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
01C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Data is displayed in ASCII with a "." in the position of non-graphic characters

-X Current CPU state

```
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=010D INX  H
```

-T5 Trace 5 steps from current CPU state

```
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=010D INX  H
C0Z0M0E0I1 A=02 B=0800 D=0000 H=011A S=0100 P=010E DCR  B
C0Z0M0E0I1 A=02 B=0700 D=0000 H=011A S=0100 P=010F JNZ  0107
C0Z0M0E0I1 A=02 B=0700 D=0000 H=011A S=0100 P=0107 MOV  A,M
C0Z0M0E0I1 A=00 B=0700 D=0000 H=011A S=0100 P=0108 SUB  C*0109
```

Automatic Breakpoint

-U5 Trace without listing intermediate states

```
C0Z1M0E1I1 A=00 B=0700 D=0000 H=011A S=0100 P=0109 JNC  010D*0108
```

-X CPU state at end of U5

```
C0Z0M0E1I1 A=04 B=0600 D=0000 H=011B S=0100 P=0108 SUB  C
```

-G Run Program from current PC until completion (in real-time)

*0116 breakpoint at 116H, caused by executing RST 7 in machine code

-X CPU state at end of Program

```
C0Z1M0E1I1 A=00 B=0000 D=0000 H=0121 S=0100 P=0116 RST  07
```

-XP examine and change Program counter

```
P=0116 100
```

```
-X
C0Z1M0E1I1 A=00 B=0000 D=0000 H=0121 S=0100 P=0100 MVI  B,08
```

-T10 Trace 10 (hexadecimal) steps

first data element — current largest value — subtract for comparison A<C

```
C0Z1M0E1I1 A=00 B=0000 D=0000 H=0121 S=0100 P=0100 MVI  B,08
C0Z1M0E1I1 A=00 B=0800 D=0000 H=0121 S=0100 P=0102 MVI  C,00
C0Z1M0E1I1 A=00 B=0800 D=0000 H=0121 S=0100 P=0104 LXI  H,0119
C0Z1M0E1I1 A=00 B=0800 D=0000 H=0119 S=0100 P=0107 MOV  A,M
C0Z1M0E1I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0108 SUB  C
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0109 JNC  010D
C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=010D INX  H
C0Z0M0E0I1 A=02 B=0800 D=0000 H=011A S=0100 P=010E DCR  B
C0Z0M0E0I1 A=02 B=0700 D=0000 H=011A S=0100 P=010F JNZ  0107
C0Z0M0E0I1 A=02 B=0700 D=0000 H=011A S=0100 P=0107 MOV  A,M
C0Z0M0E0I1 A=00 B=0700 D=0000 H=011A S=0100 P=0108 SUB  C
C0Z1M0E1I1 A=00 B=0700 D=0000 H=011A S=0100 P=0109 JNC  010D
C0Z1M0E1I1 A=00 B=0700 D=0000 H=011A S=0100 P=010D INX  H
C0Z1M0E1I1 A=00 B=0700 D=0000 H=011B S=0100 P=010E DCR  B
C0Z0M0E1I1 A=00 B=0600 D=0000 H=011B S=0100 P=010F JNZ  0107
C0Z0M0E1I1 A=00 B=0600 D=0000 H=011B S=0100 P=0107 MOV  A,M*0108
```

-A109 Insert a "hot patch" into the machine code to change the JNC to JC

```
0109 JC 10D
010C
```

Program should have moved the value from A into C since A>C. Since this code was not executed, it appears that the JNC should have been a JC instruction

-G0 Stop DDT so that a version of the patched program can be saved

SAVE 1 SCAN.COM Program resides on first page, so save 1 page.

A>DDT SCAN.COM Restart DDT with the saved memory image to continue testing

```
16K DDT VER 1.0
NEXT  PC
0200 0100
```

-L100 List some code

```
0100  MVI  B,08
0102  MVI  C,00
0104  LXI  H,0119
0107  MOV  A,M
0108  SUB  C
0109  JC   010D
```

Previous Patch is present in X.COM

```
010C   MOV   C,A
010D   INX   H
010E   DCR   B
010F   JNZ   0107
0112   MOV   A,C
-XP

P=0100

-T10   Trace to see how patched version operates          Data is moved from A to C
C0Z0M0E0I0  A=00  B=0000  D=0000  H=0000  S=0100  P=0100  MVI   B,08
C0Z0M0E0I0  A=00  B=0800  D=0000  H=0000  S=0100  P=0102  MVI   C,00
C0Z0M0E0I0  A=00  B=0800  D=0000  H=0000  S=0100  P=0104  LXI   H,0119
C0Z0M0E0I0  A=00  B=0800  D=0000  H=0119  S=0100  P=0107  MOV   A,M
C0Z0M0E0I0  A=02  B=0800  D=0000  H=0119  S=0100  P=0108  SUB   C
C0Z0M0E0I1  A=02  B=0800  D=0000  H=0119  S=0100  P=0109  JC    010D
C0Z0M0E0I1  A=02  B=0800  D=0000  H=0119  S=0100  P=010C  MOV   C,A
C0Z0M0E0I1  A=02  B=0802  D=0000  H=0119  S=0100  P=010D  INX   H
C0Z0M0E0I1  A=02  B=0802  D=0000  H=011A  S=0100  P=010E  DCR   B
C0Z0M0E0I1  A=02  B=0702  D=0000  H=011A  S=0100  P=010F  JNZ   0107
C0Z0M0E0I1  A=02  B=0702  D=0000  H=011A  S=0100  P=0107  MOV   A,M
C0Z0M0E0I1  A=00  B=0702  D=0000  H=011A  S=0100  P=0108  SUB   C
C1Z0M1E0I0  A=FE  B=0702  D=0000  H=011A  S=0100  P=0109  JC    010D
C1Z0M1E0I0  A=FE  B=0702  D=0000  H=011A  S=0100  P=010D  INX   H
C1Z0M1E0I0  A=FE  B=0702  D=0000  H=011B  S=0100  P=010E  DCR   B
C1Z0M0E1I1  A=FE  B=0602  D=0000  H=011B  S=0100  P=010F  JNZ   0107*0107
-X                                                  breakpoint after 16 steps

C1Z0M0E1I1  A=FE  B=0602  D=0000  H=011B  S=0100  P=0107  MOV   A,M
-G,108   Run from current PC and breakpoint at 108H

*0108
-X                  next data item

C1Z0M0E1I1  A=04  B=0602  D=0000  H=011B  S=0100  P=0108  SUB   C
-T                  Single Step for a few cycles

C1Z0M0E1I1  A=04  B=0602  D=0000  H=011B  S=0100  P=0108  SUB   C*0109
-T

C0Z0M0E0I1  A=02  B=0602  D=0000  H=011B  S=0100  P=0109  JC    010D*010C
-X

C0Z0M0E0I1  A=02  B=0602  D=0000  H=011B  S=0100  P=010C  MOV   C,A
-G    Run to completion
*0116
-X

C0Z1M0E1I1  A=03  B=0003  D=0000  H=0121  S=0100  P=0116  RST   07
-S121   look at the value of "LARGE"
0121 03   Wrong Value!
```

```
0122 00
0123 22
0124 21
0125 00
0126 02
0127 7E .                End of the S Command
-L100
0100  MVI  B,03
0102  MVI  C,00
0104  LXI  H,0119
0107  MOV  A,M
0108  SUB  C
0109  JC   010D
010C  MOV  C,A
010D  INX  H
010E  DCR  B
010F  JNZ  0107
0112  MOV  A,C           Review the Code
-L
0113  STA  0121
0116  RST  07
0117  NOP
0118  NOP
0119  STAX B
011A  NOP
011B  INR  B
011C  INX  B
011D  DCR  B
011E  MVI  B,01
0120  DCR  B
-XP
P=0116 100     Reset the PC
-T     Single step, and watch data values.
C0Z1M0E1I1 A=03 B=0003 D=0000 H=0121 S=0100 P=0100 MVI  B,03*0102
-T
C0Z1M0E1I1 A=03 B=0803 D=0000 H=0121 S=0100 P=0102 MVI  C,00*0104
-T
                   count set
                      largest set
C0Z1M0E1I1 A=03 B=0800 D=0000 H=0121 S=0100 P=0104 LXI  H,0119*0107
-T
                                 base address of data set
C0Z1M0E1I1 A=03 B=0800 D=0000 H=0119 S=0100 P=0107 MOV  A,M*0108
```

```
-T
                   first data item brought to A
C0Z1M0E1I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0108 SUB  C*0109
-T

C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=0109 JC   010D*010C
-T

C0Z0M0E0I1 A=02 B=0800 D=0000 H=0119 S=0100 P=010C MOV  C,A*010D
-T                      first data item moved to C correctly

C0Z0M0E0I1 A=02 B=0802 D=0000 H=0119 S=0100 P=010D INX  H*010E
-T

C0Z0M0E0I1 A=02 B=0802 D=0000 H=011A S=0100 P=010E DCR  B*010F
-T

C0Z0M0E0I1 A=02 B=0702 D=0000 H=011A S=0100 P=010F JNZ  0107*0107
-T

C0Z0M0E0I1 A=02 B=0702 D=0000 H=011A S=0100 P=0107 MOV  A,M*0108
-T                   second data item brought to A

C0Z0M0E0I1 A=00 B=0702 D=0000 H=011A S=0100 P=0108 SUB  C*0109
-T                   subtract destroys data value which was loaded!!!

C1Z0M1E0I0 A=FE B=0702 D=0000 H=011A S=0100 P=0109 JC   010D*010D
-T

C1Z0M1E0I0 A=FE B=0702 D=0000 H=011A S=0100 P=010D INX  H*010E
-L100

0100  MVI  B,08
0102  MVI  C,00
0104  LXI  H,0119
0107  MOV  A,M
0108  SUB  C      <--- This should have been a CMP so that register A
0109  JC   010D        would not be destroyed.
010C  MOV  C,A
010D  INX  H
010E  DCR  B
010F  JNZ  0107
0112  MOV  A,C
-A108

0108  CMP C           hot patch at 108H changes SUB to CMP

0109

-G0      stop DDT for SAVE
```

```
SAVE 1 SCAN.COM          Save memory image

A>DDT SCAN.COM           Restart DDT

16K DDT VER 1.0
NEXT  PC
0200 0100
-XP

P=0100

-L116

0116  RST  07
0117  NOP
0118  NOP          Look at code to see if it was properly Loaded
0119  STAX B       (long typeout aborted with rubout)
011A  NOP
- (rubout)

-G,116     Run from 100H to completion

*0116
-XC     Look at Carry (accidental typo)

C1

-X      Look at CPU state
C1Z1M0E1I1 A=06 B=0006 D=0000 H=0121 S=0100 P=0116 RST  07
-S121   Look at "Large" - it appears to be correct.
0121 06

0122 00

0123 22 .

-G0     stop DDT.

ED SCAN.ASM    Re-edit the source program, and make both changes

*NSUB
*0LT
        SUB  ctl-z  C          ;LARGER VALUE IN C?
*SSUB↑ZCMP↑Z0LT
        CMP          C          ;LARGER VALUE IN C?
*
        JNC          NFOUND     ;JUMP IF LARGER VALUE NOT FOUND
*SNC↑ZC↑Z0LT
        JC           NFOUND     ;JUMP IF LARGER VALUE NOT FOUND
*E
```

```
ASM SCAN.AAZ
```
Re-assemble, selecting source from disk A
hex to disk A
Print to Z (selects no print file)

```
CP/M ASSEMBLER - VER 1.0

0122
002H USE FACTOR
END OF ASSEMBLY

DDT SCAN.HEX
```
Re-run debugger to check changes

```
16K DDT VER 1.0
NEXT  PC
0121 0000
-L116

0116  JMP  0000
```
check to ensure end is still at 116H

```
0119  STAX B
011A  NOP
011B  INR  B
```
- (rubout)

```
-G100,116
```
Go from beginning with breakpoint at end

```
*0116
```
breakpoint reached

```
-D121
```
Look at "LARGE" — correct value computed

```
0121 06 00 22 21 00 02 7E EB 77 13 23 EB 0B 78 B1 .."!..^.W.#..X.
0130 C2 27 01 C3 03 29 00 00 00 00 00 00 00 00 00 00 .'...)..........
0140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

- (rubout) aborts long typeout

```
-G0
```
stop DDT, debug session complete

CP/M ASSEMBLER (ASM)

USER'S GUIDE

Table of Contents

CP/M Assembler User's Guide

1. INTRODUCTION.

The CP/M assembler reads assembly language source files from the diskette, and produces 8080 machine language in Intel hex format. The CP/M assembler is initiated by typing

ASM filename

or

ASM filename.parms

In both cases, the assembler assumes there is a file on the diskette with the name

filename.ASM

which contains an 8080 assembly language source file. The first and second forms shown above differ only in that the second form allows parameters to be passed to the assembler to control source file access and hex and print file destinations.

In either case, the CP/M assembler loads, and prints the message

CP/M ASSEMBLER VER n.n

where n.n is the current version number. In the case of the first command, the assembler reads the source file with assumed file type "ASM" and creates two output files

filename.HEX

and

filename.PRN

the "HEX" file contains the machine code corresponding to the original program in Intel hex format, and the "PRN" file contains an annotated listing showing generated machine code, error flags, and source lines. If errors occur during translation, they will be listed in the PRN file as well as at the console

The second command form can be used to redirect input and output files from their defaults. In this case, the "parms" portion of the command is a three letter group which specifies the origin of the source file, the destination of the hex file, and the destination of the print file. The form is

filename.p1p2p3

where p1, p2, and p3 are single letters

p1: A,B, ..., Y designates the disk name which contains

		the source file
p2:	A,B, ..., Y	designates the disk name which will receive the hex file
	Z	skips the generation of the hex file
p3:	A,B, ..., Y	designates the disk name which will receive the print file
	X	places the listing at the console
	Z	skips generation of the print file

Thus, the command

```
ASM X.AAA
```

indicates that the source file (X.ASM) is to be taken from disk A, and that the hex (X.HEX) and print (X.PRN) files are to be created also on disk A. This form of the command is implied if the assembler is run from disk A. That is, given that the operator is currently addressing disk A, the above command is equivalent to

```
ASM X
```

The command

```
ASM X.ABX
```

indicates that the source file is to be taken from disk A, the hex file is placed on disk B, and the listing file is to be sent to the console. The command

```
ASM X.BZZ
```

takes the source file from disk B, and skips the generation of the hex and print files (this command is useful for fast execution of the assembler to check program syntax).

The source program format is compatible with both the Intel 8080 assembler (macros are not currently implemented in the CP/M assembler, however), as well as the Processor Technology Software Package #1 assembler. That is, the CP/M assembler accepts source programs written in either format. There are certain extensions in the CP/M assembler which make it somewhat easier to use. These extensions are described below.

2. PROGRAM FORMAT.

An assembly language program acceptable as input to the assembler consists of a sequence of statements of the form

```
line#    label    operation    operand    ;comment
```

where any or all of the fields may be present in a particular instance. Each

-embly language statement is terminated with a carriage return and line feed (the line feed is inserted automatically by the ED program), or with the character "!" which is a treated as an end-of-line by the assembler (thus, multiple assembly language statements can be written on the same physical line if separated by exclaim symbols).

The line# is an optional decimal integer value representing the source program line number, which is allowed on any source line to maintain compatibility with the Processor Technology format. In general, these line numbers will be inserted if a line-oriented editor is used to construct the original program, and thus ASM ignores this field if present.

The label field takes the form

```
        identifier
or
        identifier:
```

and is optional, except where noted in particular statement types. The identifier is a sequence of alphanumeric characters (alphabetics and numbers), where the first character is alphabetic. Identifiers can be freely used by the programmer to label elements such as program steps and assembler directives, but cannot exceed 16 characters in length. All characters are significant in an identifier, except for the embedded dollar symbol ($) which can be used to improve readability of the name. Further, all lower case alphabetics become are treated as if they were upper case. Note that the ":" following the identifier in a label is optional (to maintain compatibility between Intel and Processor Technology). Thus, the following are all valid instances of labels

```
x        xy        long$name
x:       yxl:      longer$named$data:
XlY2     Xlx2      x234$5678$9012$3456:
```

The operation field contains either an assembler directive, or pseudo operation, or an 8080 machine operation code. The pseudo operations and machine operation codes are described below.

The operand field of the statement, in general, contains an expression formed out of constants and labels, along with arithmetic and logical operations on these elements. Again, the complete details of properly formed expressions are given below.

The comment field contains arbitrary characters following the ";" symbol until the next real or logical end-of-line. These characters are read, listed, and otherwise ignored by the assembler. In order to maintain compatability with the Processor Technology assembler, the CP/M assembler also treat statements which begin with a "*" in column one as comment statements, which are listed and ignored in the assembly process. Note that the Processor

Technology assembler has the side effect in its operation of ignoring the characters after the operand field has been scanned. This causes an ambiguous situation when attempting to be compatible with Intel's language, since arbitrary expressions are allowed in this case. Hence, programs which use this side effect to introduce comments, must be edited to place a ";" before these fields in order to assemble correctly.

The assembly language program is formulated as a sequence of statements of the above form, terminated optionally by an END statement. All statements following the END are ignored by the assembler.

3. FORMING THE OPERAND.

In order to completely describe the operation codes and pseudo operations, it is necessary to first present the form of the operand field, since it is used in nearly all statements. Expressions in the operand field consist of simple operands (labels, constants, and reserved words), combined in properly formed subexpressions by arithmetic and logical operators. The expression computation is carried out by the assembler as the assembly proceeds. Each expression must produce a 16-bit value during the assembly. Further, the number of significant digits in the result must not exceed the intended use. That is, if an expression is to be used in a byte move immediate instruction, then the most significant 8 bits of the expression must be zero. The restrictions on the expression significance is given with the individual instructions.

3.1. Labels.

As discussed above, a label is an identifier which occurs on a particular statement. In general, the label is given a value determined by the type of statement which it precedes. If the label occurs on a statement which generates machine code or reserves memory space (e.g, a MOV instruction, or a DS pseudo operation), then the label is given the value of the program address which it labels. If the label precedes an EQU or SET, then the label is given the value which results from evaluating the operand field. Except for the SET statement, an identifier can label only one statement.

When a label appears in the operand field, its value is substituted by the assembler. This value can then be combined with other operands and operators to form the operand field for a particular instruction.

3.2. Numeric Constants.

A numeric constant is a 16-bit value in one of several bases. The base, called the radix of the constant, is denoted by a trailing radix indicator. The radix indicators are

```
B     binary constant (base 2)
O     octal constant (base 8)
```

Q	octal constant (base 8)
D	decimal constant (base 1Ø)
H	hexadecimal constant (base 16)

Q is an alternate radix indicator for octal numbers since the letter O is easily confused with the digit Ø. Any numeric constant which does not terminate with a radix indicator is assumed to be a decimal constant.

A constant is thus composed as a sequence of digits, followed by an optional radix indicator, where the digits are in the appropriate range for the radix. That is binary constants must be composed of Ø and 1 digits, octal constants can contain digits in the range Ø - 7, while decimal constants contain decimal digits. Hexadecimal constants contain decimal digits as well as hexadecimal digits A (1ØD), B (11D), C (12D), D (13D), E (14D), and F (15D). Note that the leading digit of a hexadecimal constant must be a decimal digit in order to avoid confusing a hexadecimal constant with an identifier (a leading Ø will always suffice). A constant composed in this manner must evaluate to a binary number which can be contained within a 16-bit counter, otherwise it is truncated on the right by the assembler. Similar to identifiers, imbedded "$" are allowed within constants to improve their readability. Finally, the radix indicator is translated to upper case if a lower case letter is encountered. The following are all valid instances of numeric constants

1234	1234D	11ØØB	1111$ØØØØ$1111$ØØØØB
1234H	ØFFEH	3377O	33$77$22Q
3377o	Øfe3h	1234d	Øffffh

3.3. Reserved Words.

There are several reserved character sequences which have predefined meanings in the operand field of a statement. The names of 8Ø8Ø registers are given below, which, when encountered, produce the value shown to the right

A	7
B	Ø
C	1
D	2
E	3
H	4
L	5
M	6
SP	6
PSW	6

(again, lower case names have the same values as their upper case equivalents). Machine instructions can also be used in the operand field, and evaluate to their internal codes. In the case of instructions which require operands, where the specific operand becomes a part of the binary bit pattern

of the instruction (e.g, MOV A,B), the value of the instruction (in this case MOV) is the bit pattern of the instruction with zeroes in the optional fields (e.g, MOV produces 40H).

When the symbol "$" occurs in the operand field (not imbedded within identifiers and numeric constants) its value becomes the address of the next instruction to generate, not including the instruction contained withing the current logical line.

3.4. String Constants.

String constants represent sequences of ASCII characters, and are represented by enclosing the characters within apostrophe symbols (´). All strings must be fully contained within the current physical line (thus allowing "!" symbols within strings), and must not exceed 64 characters in length. The apostrophe character itself can be included within a string by representing it as a double apostrophe (the two keystrokes ´´), which becomes a single apostrophe when read by the assembler. In most cases, the string length is restricted to either one or two characters (the DB pseudo operation is an exception), in which case the string becomes an 8 or 16 bit value, respectively. Two character strings become a 16-bit constant, with the second character as the low order byte, and the first character as the high order byte.

The value of a character is its corresponding ASCII code. There is no case translation within strings, and thus both upper and lower case characters can be represented. Note however, that only graphic (printing) ASCII characters are allowed within strings. Valid strings are

```
´A´     ´AB´     ´ab´     ´c´
´´´´    ´a´´´    ´´´´´´   ´´´"´
´Walla Walla Wash.´
´She said ´´Hello´´ to me.´
´I said "Hello" to her.´
```

3.5. Arithmetic and Logical Operators.

The operands described above can be combined in normal algebraic notation using any combination of properly formed operands, operators, and parenthesized expressions. The operators recognized in the operand field are

a + b	unsigned arithmetic sum of a and b
a - b	unsigned arithmetic difference between a and b
+ b	unary plus (produces b)
- b	unary minus (identical to 0 - b)
a * b	unsigned magnitude multiplication of a and b
a / b	unsigned magnitude division of a by b
a MOD b	remainder after a / b
NOT b	logical inverse of b (all 0´s become 1´s, 1´s become 0´s), where b is considered a 16-bit value

a AND b	bit-by-bit logical and of a and b
a OR b	bit-by-bit logical or of a and b
a XOR b	bit-by-bit logicl exclusive or of a and b
a SHL b	the value which results from shifting a to the left by an amount b, with zero fill
a SHR b	the value which results from shifting a to the right by an amount b, with zero fill

In each case, a and b represent simple operands (labels, numeric constants, reserved words, and one or two character strings), or fully enclosed parenthesized subexpressions such as

```
10+20        10h+37Q      L1 /3     (L2+4) SHR 3
('a' and 5fh) + '0'       ('B'+B) OR (PSW+M)
(1+(2+c)) shr (A-(B+1))
```

Note that all computations are performed at assembly time as 16-bit unsigned operations. Thus, -1 is computed as 0-1 which results in the value 0ffffh (i.e., all 1's). The resulting expression must fit the operation code in which it is used. If, for example, the expression is used in a ADI (add immediate) instruction, then the high order eight bits of the expression must be zero. As a result, the operation "ADI -1" produces an error message (-1 becomes 0ffffh which cannot be represented as an 8 bit value), while "ADI (-1) AND 0FFH" is accepted by the assembler since the "AND" operation zeroes the high order bits of the expression.

3.6. Precedence of Operators.

As a convenience to the programmer, ASM assumes that operators have a relative precedence of application which allows the programmer to write expressions without nested levels of parentheses. The resulting expression has assumed parentheses which are defined by the relative precedence. The order of application of operators in unparenthesize expressions is listed below. Operators listed first have highest precedence (they are applied first in an unparenthesized expression), while operators listed last have lowest precedence. Operators listed on the same line have equal precedence, and are applied from left to right as they are encountered in an expression

```
* / MOD SHL SHR
    - +
    NOT
    AND
   OR XOR
```

Thus, the expressions shown to the left below are interpreted by the assembler as the fully parenthesize expressions shown to the right below

```
a * b + c                   (a * b) + c
a + b * c                   a + (b * c)
a MOD b * c SHL d           ((a MOD b) * c) SHL d
```

a OR b AND NOT c + d SHL e   a OR (b AND (NOT (c + (d SHL e))))

Balanced parenthesized subexpressions can always be used to override the assumed parentheses, and thus the last expression above could be rewritten to force application of operators in a different order as

(a OR b) AND (NOT c) + d SHL e

resulting in the assumed parentheses

(a OR b) AND ((NOT c) + (d SHL e))

Note that an unparenthesized expression is well-formed only if the expression which results from inserting the assumed parentheses is well-formed.

4. ASSEMBLER DIRECTIVES.

Assembler directives are used to set labels to specific values during the assmbly, perform conditional assembly, define storage areas, and specify starting addresses in the program. Each assembler directive is denoted by a "pseudo operation" which appears in the operation field of the line. The acceptable pseudo operations are

ORG	set the program or data origin
END	end program, optional start address
EQU	numeric "equate"
SET	numeric "set"
IF	begin conditional assembly
ENDIF	end of conditional assembly
DB	define data bytes
DW	define data words
DS	define data storage area

The individual pseudo operations are detailed below

4.1. The ORG directive.

The ORG statement takes the form

label   ORG   expression

where "label" is an optional program label, and expression is a 16-bit expression, consisting of operands which are defined previous to the ORG statement. The assembler begins machine code generation at the location specified in the expression. There can be any number of ORG statements within a particular program, and there are no checks to ensure that the programmer is not defining overlapping memory areas. Note that most programs written for the CP/M system begin with an ORG statement of the form

ORG 100H

which causes machine code generation to begin at the base of the CP/M transient program area. If a label is specified in the ORG statement, then the label is given the value of the expression (this label can then be used in the operand field of other statements to represent this expression).

4.2. The END directive.

The END statement is optional in an assembly language program, but if it is present it must be the last statement (all subsequent statements are ignored in the assembly). The two forms of the END directive are

```
label   END
label   END   expression
```

where the label is again optional. If the first form is used, the assembly process stops, and the default starting address of the program is taken as 0000. Otherwise, the expression is evaluated, and becomes the program starting address (this starting address is included in the last record of the Intel formatted machine code "hex" file which results from the assembly). Thus, most CP/M assembly language programs end with the statement

```
END 100H
```

resulting in the default starting address of 100H (beginning of the transient program area).

4.3. The EQU directive.

The EQU (equate) statement is used to set up synonyms for particular numeric values. the form is

```
label   EQU   expression
```

where the label must be present, and must not label any other statement. The assembler evaluates the expression, and assigns this value to the identifier given in the label field. The identifier is usually a name which describes the value in a more human-oriented manner. Further, this name is used throughout the program to "parameterize" certain functions. Suppose for example, that data received from a Teletype appears on a particular input port, and data is sent to the Teletype through the next output port in sequence. The series of equate statements could be used to define these ports for a particular hardware environment

```
TTYBASE   EQU   10H        ;BASE PORT NUMBER FOR TTY
TTYIN     EQU   TTYBASE    ;TTY DATA IN
TTYOUT    EQU   TTYBASE+1;TTY DATA OUT
```

At a later point in the program, the statements which access the Teletype could appear as

```
        IN      TTYIN      ;READ TTY DATA TO REG-A
        ...
        OUT     TTYOUT     ;WRITE DATA TO TTY FROM REG-A
```

making the program more readable than if the absolute i/o ports had been used. Further, if the hardware environment is redefined to start the Teletype communications ports at 7FH instead of 10H, the first statement need only be changed to

```
TTYBASE   EQU   7FH      ;BASE PORT NUMBER FOR TTY
```

and the program can be reassembled without changing any other statements.

4.4. The SET Directive.

The SET statement is similar to the EQU, taking the form

```
label   SET   expression
```

except that the label can occur on other SET statements within the program. The expression is evaluated and becomes the current value associated with the label. Thus, the EQU statement defines a label with a single value, while the SET statement defines a value which is valid from the current SET statement to the point where the label occurs on the next SET statement. The use of the SET is similar to the EQU statement, but is used most often in controlling conditional assembly.

4.5. The IF and ENDIF directives.

The IF and ENDIF statements define a range of assembly language statements which are to be included or excluded during the assembly process. The form is

```
IF   expression
statement#1
statement#2
   ...
statement#n
ENDIF
```

Upon encountering the IF statement, the assembler evaluates the expression following the IF (all operands in the expression must be defined ahead of the IF statement). If the expression evaluates to a non-zero value, then statement#1 through statement#n are assembled; if the expression evaluates to zero, then the statements are listed but not assembled. Conditional assembly is often used to write a single "generic" program which includes a number of possible run-time environments, with only a few specific portions of the program selected for any particular assembly. The following program segments for example, might be part of a program which communicates with either a Teletype or a CRT console (but not both) by selecting a particular value for TTY before the assembly begins

```
TRUE     EQU   0FFFFH       ;DEFINE VALUE OF TRUE
FALSE    EQU   NOT TRUE     ;DEFINE VALUE OF FALSE
;
TTY      EQU   TRUE         ;TRUE IF TTY, FALSE IF CRT
;
TTYBASE  EQU   10H          ;BASE OF TTY I/O PORTS
CRTBASE  EQU   20H          ;BASE OF CRT I/O PORTS
         IF    TTY          ;ASSEMBLE RELATIVE TO TTYBASE
CONIN    EQU   TTYBASE      ;CONSOLE INPUT
CONOUT   EQU   TTYBASE+1    ;CONSOLE OUTPUT
         ENDIF
;
         IF    NOT TTY      ;ASSEMBLE RELATIVE TO CRTBASE
CONIN    EQU   CRTBASE      ;CONSOLE INPUT
CONOUT   EQU   CRTBASE+1    ;CONSOLE OUTPUT
         ENDIF
         ...
         IN     CONIN       ;READ CONSOLE DATA
         ...
         OUT    CONOUT      ;WRITE CONSOLE DATA
```

In this case, the program would assemble for an environment where a Teletype is connected, based at port 10H. The statement defining TTY could be changed to

```
TTY      EQU    FALSE
```

and, in this case, the program would assemble for a CRT based at port 20H.

4.6. The DB Directive.

The DB directive allows the programmer to define initialize storage areas in single precision (byte) format. The statement form is

```
label   DB   e#1, e#2, ..., e#n
```

where e#1 through e#n are either expressions which evaluate to 8-bit values (the high order eight bits must be zero), or are ASCII strings of length no greater than 64 characters. There is no practical restriction on the number of expressions included on a single source line. The expressions are evaluated and placed sequentially into the machine code file following the last program address generated by the assembler. String characters are similarly placed into memory starting with the first character and ending with the last character. Strings of length greater than two characters cannot be used as operands in more complicated expressions (i.e., they must stand alone between the commas). Note that ASCII characters are always placed in memory with the parity bit reset (0). Further, recall that there is no translation from lower to upper case within strings. The optional label can be used to reference the data area throughout the remainder of the program. Examples of

valid DB statements are

```
data:   DB   0,1,2,3,4,5
        DB   data and 0ffh,5,377Q,1+2+3+4
signon: DB   'please type your name',cr,lf,0
        DB   'AB' SHR 8, 'C', 'DE' AND 7FH
```

4.7. The DW Directive.

The DW statement is similar to the DB statement except double precision (two byte) words of storage are initialized. The form is

```
label   DW   e#1, e#2, ..., e#n
```

where e#1 through e#n are expressions which evaluate to 16-bit results. Note that ASCII strings of length one or two characters are allowed, but strings longer than two characters disallowed. In all cases, the data storage is consistent with the 8080 processor: the least significant byte of the expression is stored forst in memory, followed by the most significant byte. Examples are

```
doub:   DW   0ffefh,doub+4,signon-$,255+255
        DW   'a', 5, 'ab', 'CD', 6 shl 8 or 11b
```

4.8. The DS Directive.

The DS statement is used to reserve an area of uninitialized memory, and takes the form

```
label   DS   expression
```

where the label is optional. The assembler begins subsequent code generation after the area reserved by the DS. Thus, the DS statement given above has exactly the same effect as the statement

```
label:  EQU  $    ;LABEL VALUE IS CURRENT CODE LOCATION
        ORG  $+expression   ;MOVE PAST RESERVED AREA
```

5. OPERATION CODES.

Assembly language operation codes form the principal part of assembly language programs, and form the operation field of the instruction. In general, ASM accepts all the standard mnemonics for the Intel 8080 microcomputer, which are given in detail in the Intel manual "8080 Assembly Language Programming Manual." Labels are optional on each input line and, if included, take the value of the instruction address immediately before the instruction is issued. The individual operators are listed breifly in the

following sections for completeness, although it is understood that the Intel manuals should be referenced for exact operator details. In each case,

e3	represents a 3-bit value in the range 0-7 which can be one of the predefined registers A, B, C, D, E, H, L, M, SP, or PSW.
e8	represents an 8-bit value in the range 0-255
e16	represents a 16-bit value in the range 0-65535

which can themselves be formed from an arbitrary combination of operands and operators. In some cases, the operands are restricted to particular values within the allowable range, such as the PUSH instruction. These cases will be noted as they are encountered.

In the sections which follow, each operation codes is listed in its most general form, along with a specific example, with a short explanation and special restrictions.

5.1. Jumps, Calls, and Returns.

The Jump, Call, and Return instructions allow several different forms which test the condition flags set in the 8080 microcomputer CPU. The forms are

JMP e16	JMP L1	Jump unconditionally to label
JNZ e16	JMP L2	Jump on non zero condition to label
JZ e16	JMP 100H	Jump on zero condition to label
JNC e16	JNC L1+4	Jump no carry to label
JC e16	JC L3	Jump on carry to label
JPO e16	JPO $+8	Jump on parity odd to label
JPE e16	JPE L4	Jump on even parity to label
JP e16	JP GAMMA	Jump on positive result to label
JM e16	JM a1	Jump on minus to label
CALL e16	CALL S1	Call subroutine unconditionally
CNZ e16	CNZ S2	Call subroutine if non zero flag
CZ e16	CZ 100H	Call subroutine on zero flag
CNC e16	CNC S1+4	Call subroutine if no carry set
CC e16	CC S3	Call subroutine if carry set
CPO e16	CPO $+8	Call subroutine if parity odd
CPE e16	CPE S4	Call subroutine if parity even
CP e16	CP GAMMA	Call subroutine if positive result
CM e16	CM b1$c2	Call subroutine if minus flag
RST e3	RST 0	Programmed "restart", equivalent to CALL 8*e3, except one byte call

RET	Return from subroutine
RNZ	Return if non zero flag set
RZ	Return if zero flag set
RNC	Return if no carry
RC	Return if carry flag set
RPO	Return if parity is odd
RPE	Return if parity is even
RP	Return if positive result
RM	Return if minus flag is set

5.2. Immediate Operand Instructions.

Several instructions are available which load single or double precision registers, or single precision memory cells, with constant values, along with instructions which perform immediate arithmetic or logical operations on the accumulator (register A).

MVI e3,e8	MVI B,255	Move immediate data to register A, B, C, D, E, H, L, or M (memory)
ADI e8	ADI 1	Add immediate operand to A without carry
ACI e8	ACI 0FFH	Add immediate operand to A with carry
SUI e8	SUI L + 3	Subtract from A without borrow (carry)
SBI e8	SBI L AND 11B	Subtract from A with borrow (carry)
ANI e8	ANI $ AND 7FH	Logical "and" A with immediate data
XRI e8	XRI 1111$0000B	"Exclusive or" A with immediate data
ORI e8	ORI L AND 1+1	Logical "or" A with immediate data
CPI e8	CPI ´a´	Compare A with immediate data (same as SUI except register A not changed)
LXI e3,e16	LXI B,100H	Load extended immediate to register pair (e3 must be equivalent to B,D,H, or SP)

5.3. Increment and Decrement Instructions.

Instructions are provided in the 8080 repetoire for incrementing or decrementing single and double precision registers. The instructions are

INR e3	INR E	Single precision increment register (e3 produces one of A, B, C, D, E, H, L, M)
DCR e3	DCR A	Single precision decrement register (e3 produces one of A, B, C, D, E, H, L, M)
INX e3	INX SP	Double precision increment register pair (e3 must be equivalent to B,D,H, or SP)
DCX e3	DCX B	Double precision decrement register pair (e3 must be equivalent to B,D,H, or SP)

5.4. Data Movement Instructions.

Instructions which move data from memory to the CPU and from CPU to memory are given below

MOV e3,e3	MOV A,B	Move data to leftmost element from rightmost element (e3 produces one of A,B,C D,E,H,L, or M). MOV M,M is disallowed
LDAX e3	LDAX B	Load register A from computed address (e3 must produce either B or D)
STAX e3	STAX D	Store register A to computed address (e3 must produce either B or D)
LHLD e16	LHLD L1	Load HL direct from location e16 (double precision load to H and L)
SHLD e16	SHLD L5+x	Store HL direct to location e16 (double precision store from H and L to memory)
LDA e16	LDA Gamma	Load register A from address e16
STA e16	STA X3-5	Store register A into memory at e16
POP e3	POP PSW	Load register pair from stack, set SP (e3 must produce one of B, D, H, or PSW)
PUSH e3	PUSH B	Store register pair into stack, set SP (e3 must produce one of B, D, H, or PSW)
IN e8	IN Ø	Load register A with data from port e8
OUT e8	OUT 255	Send data from register A to port e8
XTHL		Exchange data from top of stack with HL
PCHL		Fill program counter with data from HL
SPHL		Fill stack pointer with data from HL
XCHG		Exchange DE pair with HL pair

5.5. Arithmetic Logic Unit Operations.

Instructions which act upon the single precision accumulator to perform arithmetic and logic operations are

ADD e3	ADD B	Add register given by e3 to accumulator without carry (e3 must produce one of A, B, C, D, E, H, or L)
ADC e3	ADC L	Add register to A with carry, e3 as above
SUB e3	SUB H	Subtract reg e3 from A without carry, e3 is defined as above
SBB e3	SBB 2	Subtract register e3 from A with carry, e3 defined as above
ANA e3	ANA 1+1	Logical "and" reg with A, e3 as above
XRA e3	XRA A	"Exclusive or" with A, e3 as above
ORA e3	ORA B	Logical "or" with A, e3 defined as above
CMP e3	CMP H	Compare register with A, e3 as above
DAA		Decimal adjust register A based upon last arithmetic logic unit operation
CMA		Complement the bits in register A
STC		Set the carry flag to 1

CMC		Complement the carry flag
RLC		Rotate bits left, (re)set carry as a side effect (high order A bit becomes carry)
RRC		Rotate bits right, (re)set carry as side effect (low order A bit becomes carry)
RAL		Rotate carry/A register to left (carry is involved in the rotate)
RAR		Rotate carry/A register to right (carry is involved in the rotate)
DAD e3	DAD B	Double precision add register pair e3 to HL (e3 must produce B, D, H, or SP)

5.6. Control Instructions.

The four remaining instructions are categorized as control instructions, and are listed below

HLT	Halt the 8080 processor
DI	Disable the interrupt system
EI	Enable the interrupt system
NOP	No operation

6. ERROR MESSAGES.

When errors occur within the assembly language program, they are listed as single character flags in the leftmost position of the source listing. The line in error is also echoed at the console so that the source listing need not be examined to determine if errors are present. The error codes are

D	Data error: element in data statement cannot be placed in the specified data area
E	Expression error: expression is ill-formed and cannot be computed at assembly time
L	Label error: label cannot appear in this context (may be duplicate label)
N	Not implemented: features which will appear in future ASM versions (e.g., macros) are recognized, but flagged in this version)
O	Overflow: expression is too complicated (i.e., too many pending operators) to computed, simplify it
P	Phase error: label does not have the same value on two subsequent passes through the program

R	Register error: the value specified as a register is not compatible with the operation code
V	Value error: operand encountered in expression is improperly formed

Several error message are printed which are due to terminal error conditions

NO SOURCE FILE PRESENT	The file specified in the ASM command does not exist on disk
NO DIRECTORY SPACE	The disk directory is full, erase files which are not needed, and retry
SOURCE FILE NAME ERROR	Improperly formed ASM file name (e.g., it is specified with "?" fields)
SOURCE FILE READ ERROR	Source file cannot be read properly by the assembler, execute a TYPE to determine the point of error
OUTPUT FILE WRITE ERROR	Output files cannot be written properly, most likely cause is a full disk, erase and retry
CANNOT CLOSE FILE	Output file cannot be closed, check to see if disk is write protected

7. A SAMPLE SESSION.

The following session shows interaction with the assembler and debugger in the development of a simple assembly language program.

```
ASM SORT    assemble SORT.ASM

CP/M ASSEMBLER - VER 1.0

015C next free address
003H USE FACTOR % of table used 00 TO FF (hexadecimal)
END OF ASSEMBLY

DIR SORT.*

SORT     .ASM source file
SORT      BAK backup from last edit
SORT      PRN print file (contains tab characters)
SORT      HEX machine code file
A>TYPE SORT.PRN

                              source line

machine code location ;          SORT PROGRAM IN CP/M ASSEMBLY LANGUAGE
                     ;           START AT THE BEGINNING OF THE TRANSIENT PROGRAM AR
 0100                .           ORG      100H
     generated machine code
 0100 214601          SORT:      LXI      H,SW      ;ADDRESS SWITCH TOGGLE
 0103 3601                       MVI      M,1       ;SET TO 1 FOR FIRST ITERATION
 0105 214701                     LXI      H,I       ;ADDRESS INDEX
 0108 3600                       MVI      M,0       ;I = 0
                     ;
                     ;           COMPARE I WITH ARRAY SIZE
 010A 7E              COMP:      MOV      A,M       ;A REGISTER = I
 010B FE09                       CPI      N-1       ;CY SET IF I < (N-1)
 010D D21901                     JNC      CONT      ;CONTINUE IF I <= (N-2)
                     ;
                     ;           END OF ONE PASS THROUGH DATA
 0110 214601                     LXI      H,SW      ;CHECK FOR ZERO SWITCHES
 0113 7EB7C20001                 MOV A,M! ORA A! JNZ SORT ;END OF SORT IF SW=0
                     ;
 0118 FF                         RST 7              ;GO TO THE DEBUGGER INSTEAD OF 

                     ; truncated
                     ;           CONTINUE THIS PASS
                     ;           ADDRESSING I, SO LOAD AV(I) INTO REGISTERS
 0119 5F16002148CONT:            MOV E,A! MVI D,0! LXI H,AV! DAD D! DAD D
 0121 4E792346       .           MOV C,M! MOV A,C! INX H! MOV B,M
                     ;           LOW ORDER BYTE IN A AND C, HIGH ORDER BYTE IN B
                     ;
                     ;           MOV H AND L TO ADDRESS AV(I+1)
 0125 23                         INX      H
                     ;
                     ;           COMPARE VALUE WITH REGS CONTAINING AV(I)
 0126 965778239E                 SUB M! MOV D,A! MOV A,B! INX H! SBB M    ;SUBTRACT
                     ;
                     ;           BORROW SET IF AV(I+1) > AV(I)
 012B DA3F01                     JC       INCI      ;SKIP IF IN PROPER ORDER
                     ;
                     ;           CHECK FOR EQUAL VALUES
 012E B2CA3F01                   ORA D! JZ INCI ;SKIP IF AV(I) = AV(I+1)
```

```
0132 56702B5E            MOV D,M! MOV M,B! DCX H! MOV E,M
0136 712B722B73          MOV M,C! DCX H! MOV M,D! DCX H! MOV M,E
                ;
                ;        INCREMENT SWITCH COUNT
013B 21460134            LXI H,SW! INR M
                ;
                ;        INCREMENT I
013F 21470134C3INCI:     LXI H,I! INR M! JMP COMP
                ;
                ;        DATA DEFINITION SECTION
0146 00         SW:      DB      0        ;RESERVE SPACE FOR SWITCH COUNT
0147            I:       DS       1       ;SPACE FOR INDEX
0148 05006400 1EAV:      DW      5,100,30,50,20,7,1000,300,100,-32767
000A =          N        EQU     ($-AV)/2        ;COMPUTE N INSTEAD OF PRE
015C  equate value       END
A>TYPE SORT.HEX

:10010000214601360121470136007EFE09D2190140
:1001100021460017EB7C20001FF5F16002148011983
:100120000194E7923462396577823 9EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7
:07014000470134C30A01006E
:10014800050064001E00320014000700E8032C01BB
:0401580064000180BE
:0000000000
```

machine code in HEX format

```
A>DDT SORT.HEX   start debug run

16K DDT VER 1.0
NEXT  PC
015C 0000 default address (no address on END statement)
-XP

P=0000 100  Change PC to 100

-UFFFF  untrace for 65535 steps
                                                          abort with rubout
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 LXI  H,0146*0100
-T10  trace 10₁₆ steps

C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0100 LXI  H,0146
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0103 MVI  M,01
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0105 LXI  H,0147
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=0108 MVI  M,00
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=010A MOV  A,M
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010B CPI  09
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010D JNC  0119
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0110 LXI  H,0146
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0146 S=0100 P=0113 MOV  A,M
C1Z0M1E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0114 ORA  A
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0115 JNZ  0100
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0100 LXI  H,0146
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0103 MVI  M,01
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0146 S=0100 P=0105 LXI  H,0147
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=0108 MVI  M,00
C0Z0M0E0I0 A=01 B=0000 D=0000 H=0147 S=0100 P=010A MOV  A,M*010B
-A10D
                                                    stopped at 10BH
010D  JC 119  change to a jump on carry
0110
```

```
-XP
P=010B 100
```
reset program counter back to beginning of program

```
-T10
```
trace execution for 10H steps

Altered instruction

```
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0100 LXI  H,0146
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0146 S=0100 P=0103 MVI  M,01
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0146 S=0100 P=0105 LXI  H,0147
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0108 MVI  M,00
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010A MOV  A,M
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010B CPI  09
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=010D JC   0119
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=0119 MOV  E,A
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=011A MVI  D,00
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0147 S=0100 P=011C LXI  H,0148
C1Z0M1E0I0 A=00 B=0000 D=0000 H=0148 S=0100 P=011F DAD  D
C0Z0M1E0I0 A=00 B=0000 D=0000 H=0148 S=0100 P=0120 DAD  D
C0Z0M1E0I0 A=00 B=0000 D=0000 H=0148 S=0100 P=0121 MOV  C,M
C0Z0M1E0I0 A=00 B=0005 D=0000 H=0148 S=0100 P=0122 MOV  A,C
C0Z0M1E0I0 A=05 B=0005 D=0000 H=0148 S=0100 P=0123 INX  H
C0Z0M1E0I0 A=05 B=0005 D=0000 H=0149 S=0100 P=0124 MOV  B,M*0125
-L100
```

Automatic breakpoint

```
0100  LXI   H,0146
0103  MVI   M,01
0105  LXI   H,0147
0108  MVI   M,00
010A  MOV   A,M
010B  CPI   09
010D  JC    0119
0110  LXI   H,0146
0113  MOV   A,M
0114  ORA   A
0115  JNZ   0100
```
list some code from 100H

```
-L
0118  RST   07
0119  MOV   E,A
011A  MVI   D,00
011C  LXI   H,0148
```
list more

- abort list with rubout

```
-G,118
```
start program from current PC (0125H) and run in real time to 11BH

```
*0127
```
stopped with an external interrupt 7 from front panel (program was looping indefinitely)

```
-T4
```
look at looping program in trace mode

```
C0Z0M0E0I0 A=38 B=0064 D=0006 H=0156 S=0100 P=0127 MOV  D,A
C0Z0M0E0I0 A=38 B=0064 D=3806 H=0156 S=0100 P=0128 MOV  A,B
C0Z0M0E0I0 A=00 B=0064 D=3806 H=0156 S=0100 P=0129 INX  H
C0Z0M0E0I0 A=00 B=0064 D=3806 H=0157 S=0100 P=012A SBB  M*012B
-D148
```
data is sorted, but program doesn't stop.

```
0148 05 00 07 00 14 00 1E 00 ........
0150 32 00 64 00 64 00 2C 01 E8 03 01 80 00 00 00 00 2.D.D...........
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
-GØ  return to CP/M

DDT SORT.HEX   reload the memory image

16K DDT VER 1.0
NEXT  PC
015C 0000
-XP

P=0000 100   set PC to beginning of program

-L10D   list bad opcode

010D  JNC  0119
0110  LXI  H,0146
- abort list with rubout

-A10D   assemble new opcode

010D  JC 119

0110

-L100   list starting section of program

0100  LXI  H,0146
0103  MVI  M,01
0105  LXI  H,0147
0108  MVI  M,00
- abort list with rubout

-A103   change "switch" initialization to ØØ

0103  MVI M,0

0105

-^C  return to CP/M with ctl-C (GØ works as well)

SAVE 1 SORT.COM   save 1 page (256 bytes, from 100H to 1FFH) on disk in case
                  we have to reload later

A>DDT SORT.COM   restart DDT with
                 saved memory image
16K DDT VER 1.0
NEXT  PC
0200 0100 "COM" file always starts with address 100H
-G   run the program from PC=100H

*0118  programmed stop (RST7) encountered
-D148
                                  data properly sorted
0148 05 00 07 00 14 00 1E 00 ........
0150 32 00 64 00 64 00 2C 01 E8 03 01 80 00 00 00 00 2.D.D.,.........
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

-GØ  return to CP/M
```

```
ED SORT.ASM  make changes to original program
    ctl-z
*N,0^Z0TT  find next ",0"
        MVI     M,0     ;I = 0
*-  up one line in text
        LXI     H,I     ;ADDRESS INDEX
*-  up another line
        MVI     M,1     ;SET TO 1 FOR FIRST ITERATION
*KT  kill line and type next line
        LXI     H,I     ;ADDRESS INDEX
*I  insert new line
        MVI     M,0     ;ZERO SW
*T
        LXI     H,I     ;ADDRESS INDEX
*NJNC^Z0T
        JNC*T
        CONT    ;CONTINUE IF I <= (N-2)
*-2DIC^Z0LT
        JC      CONT    ;CONTINUE IF I <= (N-2)
*E
                 source from disk A
                 hex to disk A
ASM SORT.AAZ  skip prn file

CP/M ASSEMBLER - VER 1.0

015C next address to assemble
003H USE FACTOR
END OF ASSEMBLY

DDT SORT.HEX  test program changes

16K DDT VER 1.0
NEXT  PC
015C 0000
-G100

*0118
-D148
                              data sorted
0148 05 00 07 00 14 00 1E 00 ........
0150 32 00 64 00 64 00 2C 01 EB 03 01 80 00 00 00 00 2 D D.,......
0160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

- abort with rubout

-G0  return to CP/M - program checks OK.
```